X マガジン編集部監修
MSX turbo R
Technical
Hand Book
SX turbo R テクニカル・ハンドブック
アスキー出版局

MSX turbo R Technical Hand Book

アスキー出版局

本書は、扉ページと奥付と一部の図版を除いてすべて、アスキー製『日本語 TEX』によって組版処理を行ないました。msdos.sty の作者である ishii@cts.dnp.co.jp さんと、出版技術部のみなさんに感謝します。また、扉ページのイラストを快く引き受けてくれた、めるへんめーかーさん、どうもありがとう。

なお、書籍が膨大なものとなってしまうため、今回は『日本語 MSX-DOS2』と、メモリーマッパーに関する記述は省いてあります。これらに関しては、近日発売予定の『日本語 MSX-DOS2 テクニカル・ハンドブック (仮題)』で、解説する予定です。

はじめに

MSX turbo R の世界にようこそ。本書は、高速 CPU と大容量メモリーを得て、見違えるほどパワフルになった MSX パーソナルコンピューターを、極限まで使いこなすために必要な下記のような内部情報を詳しく解説したものです。

1. 内部を 16 ビット化し、これまでの MSX と比較して 10 倍以上の処理速度を発揮する高速 CPU 、R800 の性能をぎりぎりまで引き出すテクニック。
2. MSX turbo R に標準搭載された PCM 音源と、FM 音源を使いこなすための情報とテクニック。
3. MSX を使いこなすために必須の SLOT 機構のしくみと、取扱方法。
4. 日本語を取り扱うソフトウェアの開発に必要な、漢字 BASIC の仕組み。
5. 画面表示でテクニックを発揮するための、VDP の使いこなし方法。

MSX turbo R は、従来機のアーキテクチャーを大きく変えることなく、CPU を 16 ビット化して飛躍的な高性能を実現した、はじめてのパーソナルコンピューターです。

ほかの機種では、8 ビットから 16 ビットに移行するときにアーキテクチャーをまったく変更してしまったため、8 ビットのマシンで多くの人々によって開発されたソフトウェアやノウハウは、すべて捨て去られる結果となってしまいました。

私たちは、MSX の性能を上げるために CPU を 16 ビットとすることは必要だが、そのために MSX のために開発されたソフトウェアやハードウェアの資産、またユーザーのノウハウを捨て去るようなことは、してはならないと考えました。このことを実現するためには、新しい MSX のために Z80 に上位互換な CPU が必要と考え、R800 を開発しました。そして、これまでの MSX との完全な互換性を実現するために、従来の Z80 も新開発の R800 と共に搭載した、MSX turbo R を開発しました。

MSX turbo R では、このように従来の MSX との上位互換性が理想的に保たれています。したがってユーザーは、いままでに積み上げられたソフトウェアの資産を

そのまま MSX turbo R で実行するだけで、何倍もの性能の向上を手に入れることができます[1]。また、ソフトウェアを開発するために必要な知識も、これまでのものをそのまま活用することができますが、本書で解説する若干のノウハウを利用することで、さらにマシンの性能を引き出し、群を抜くコストパフォーマンスを発揮するシステムを実現することが可能となるでしょう。

システム事業部第 1 製品統括部・統括部長　山下良蔵

[1] MSX 用の市販ソフトウェアは、R800 で実行すると速度が速くなり過ぎ互換性がとれなくなるので、自動的に Z80 が動作するため高速にならない場合があります。

目 次

図目次

表 目 次

第1章 MSX turbo R

この章は、MSX マガジン 1990 年 11 月号、1990 年 12 月号の “MSX turbo R テクニカル・アナリシス” と、“PCM 限界ギリギリ活用法” の記事を再編集したものである。

1.1 MSX turbo R のハードウェア

新開発の 16 ビット CPU “R800” を搭載したり、256 キロバイトのメイン RAM や、階層化ディレクトリーをサポートした MSX–DOS2 の標準装備など、何かと話題の多い MSX turbo R 。この注目のマシンのシステム構成はどうなっているのか、その概要を紹介する。

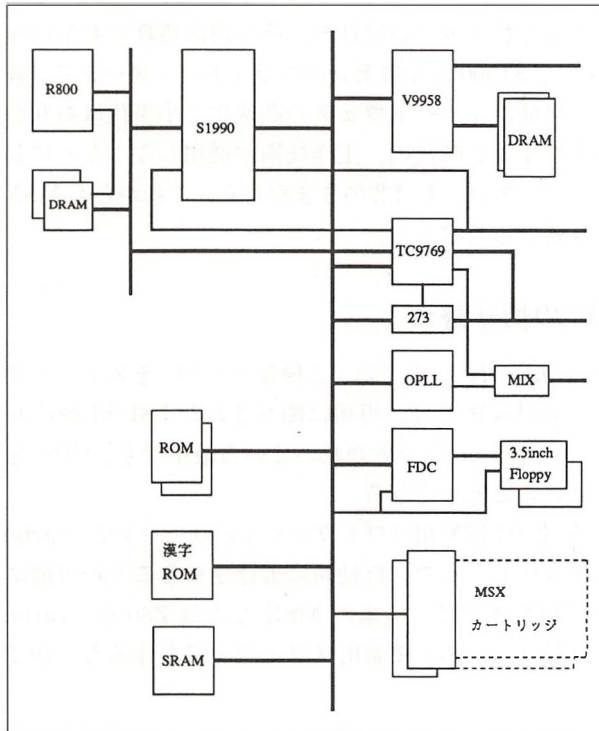
1.1.1 MSX turbo R の特徴はこれだ！

- Z80 に加え上位互換の高速 CPU “R800” を搭載することで、平均 4~5 倍、最大で 10 倍ほどのスピードを実現 (対 MSX2+比) 。
- MSX–DOS1 とともに、日本語 MSX–DOS2 と漢字ドライバーを搭載。MS–DOS コンパチブルな階層化ディレクトリーや、環境変数をサポート。
- メモリーマッパーに対応した、256 キロバイトのメイン RAM を標準で搭載。さらにスロット構成も標準化された。
- PCM の録音/再生機能を標準搭載。従来はオプション装備となっていた MSX–MUSIC も、標準装備されることになった。

1.1.2 MSX turbo R のシステム構成

MSX turbo R(以下 turbo R) のハードウェア構成は図 1.1 のとおり。従来の MSX と同じ “Z80” 互換 CPU と、新しく開発された “R800”CPU が含まれている。業界内の、“次の MSX にはザイログ社の Z280 か、日立の HD64180(どちらも Z80 互換の高速 CPU) が載るらしい” という噂に反して、何とアスキーが CPU を作ってしまったのだ。

これらのハードウェアの性能は、少し前の 16 ビット機に匹敵し、CPU の速さは V30(NEC が開発した 16 ビット CPU) なみだ。また漢字変換辞書を ROM に入れ、RAM とディスク容量を節約することは、MSX の伝統的な設計方針。最近のノート型パソコンの一部でも採用されている。turbo R のハードウェアを一言で評価すると、“みんなこれを目指してきた” といえるだろう。

図 1.1: MSX turbo R のシステム構成

turbo R のハードウェア構成を、細かい制御信号線を省略し、簡単に表わす。V9958 の出力はビデオ信号、TC9769(Z80) につながる線はキーボードとジョイスティック、273 の出力はプリンターポート、MIX の出力はオーディオ信号だ。

ハードウェア構成をもう少し詳しく説明していくと、まず、図中の **“TC9769”** は、型番から推定すると東芝製の CMOS–LSI(低消費電力のデジタル LSI の一種)。一般に **“MSX–Engine”** と呼ばれる、Z80 互換 CPU と、PSG 音源などをふくむ LSI らしい。以下、この本の中で “Z80” という表記が出てきた場合は、このチップを意味する。

その下の “273” は、プリンターを制御するためのバスバッファー、“OPLL” は FM 音源、“FDC” はフロッピーディスクコントローラーのことだ。また “SRAM” というのは、漢字辞書の学習結果を電源を切っても保存するためのメモリーだけれど、この SRAM と連文節変換辞書については、メーカーオプション機能となっている。

ところで、R800 とメイン RAM は、S1990 をとおして、バス (図では長い縦線) につながっている。たとえば、R800 が VDP を操作するときなどは、S1990 が信号を中継し、さらに必要に応じて R800 にウェイト信号を送って信号のタイミングを Z80 の信号のタイミングに合わせる操作を行なう。逆に、Z80 がメイン RAM を使うときは、S1990 と R800 が信号を中継し、メモリーマッピングを処理するわけだ。

turbo R が、こうした複雑な構成になった理由は、すべて従来のハードウェアやソフトウェアとの互換性を保つため。よくここまでやったと思う。あっぱれ。

さてこのように、turbo R の部品点数は少ないけれど、その内部処理は非常に複雑なものへと変化を遂げた。さらに、S1990 は 160 ピンのフラットパッケージで、普通の手作業によるハンダ付けは不可能だ。ハードウェアの高速化と小型化はありがたいけれど、古き良きワンボードマイコン時代の、工作技術が通用しなくなってしまった。しかし、MSX のカートリッジスロットは昔のままだから、これからも MSX はハードウェア入門の教材であり続けるだろう。

1.1.3　エレガントな CPU の切り替え

ビクターの MSX2 マシン、HC–90 と HC–95 では、2 種類の CPU をスイッチで切り替えて使っていた。ところが turbo R では、専用に開発された LSI“S1990” がシステムを管理するので、電源が入ってプログラムが動いている最中でも、CPU を切り替えてプログラムの実行を続けることができる。

このハードウェアのおかげで、従来の MSX 用ソフトウェアは Z80 モードで、turbo R 専用のソフトウェアは高速な R800 モードで、自動的に実行させることが可能になった。また、ハードウェアの種類を調べて、従来の MSX ならば Z80 を、turbo R ならば R800 を選ぶような、MSX2 / turbo R 兼用ソフトウェアも作ることができる。

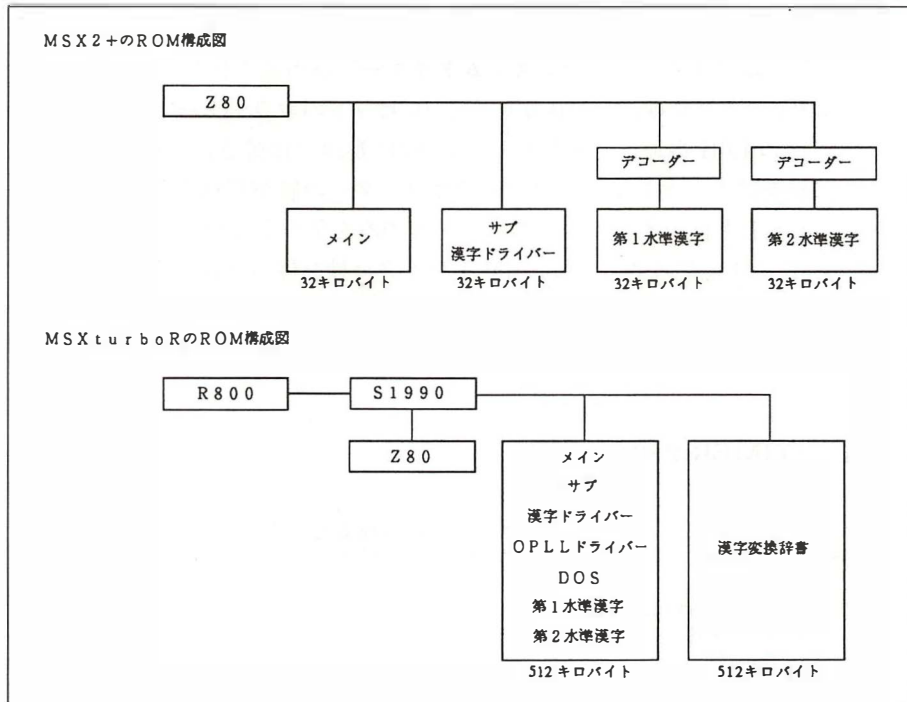
1.1.4　何でも詰め込む MSX turbo R の ROM 構成

turbo R には多くの ROM が内蔵されているはずだけれど、ふたを開けてみると ROM の数が意外に少ない。その理由が、S1990 が持つメガ ROM 制御機能だ。

MSX2+には、図 1.2 の上側のような ROM が内蔵されている。メイン ROM とサブ ROM はべつべつのスロットに接続され、漢字 ROM は I/O ポートに接続されるので、合計の容量にかかわらず、べつべつの ROM である必要があるわけだ。しかし、32 キロバイトの ROM を 4 個使うよりも、128 キロバイトの ROM を 1 個使う方が、価格も安いし、基板の面積も消費電力も小さくできる。

そこで、turbo R では、図 1.2の下側のように、1 個の 512 キロバイトの ROM にメイン、サブ、OPLL ドライバー、DOS 、第 1 水準漢字、第 2 水準漢字のすべてを詰め込んでしまった。しかし CPU と ROM の間に入っている、S1990 のメガ ROM 制御機能により、ソフトウェアからは、たとえば第 1 水準漢字 ROM は、I/O ポートの D8H 番地と D9H 番地に接続されているように見えるわけだ。

また、合計 64 キロバイトの DOS の ROM(MSX–DOS が 16 キロバイト、MSX–DOS2 が 48 キロバイト) は、スロット 3–2 の 16 キロバイトの空間に、4 バンク切り

図 1.2: MSX turbo R での ROM 構成の変化

替え方式で接続されている。

1.1.5 速さを調節するシステムタイマー

R800 が V9958(画面表示を制御する LSI) を 8 マイクロ秒以内の間隔で使おうとすると、S1990 に内蔵された VDP インターフェース回路が、自動的に R800 にウェイトをかける。これにより、CPU の処理が速すぎるために、V9958 が誤動作する心配はない。

しかし、ほかの周辺 LSI には自動的なウェイト機能がないので、ソフトウェア自身がタイミングを調整する必要がある。従来のソフトウェアの多くは、

```
EX      (SP),HL
EX      (SP),HL
```

または、

```
PUSH    HL
POP     HL
```

のような、時間がかかるけれども副作用がない命令をプログラムに埋め込んで、タイミングを調整していた。しかし、あとで説明するように、R800の命令の実行時間は不確定なので、こうした方法でタイミングを取ることは不可能だ。そこでturbo Rには、速さを調整するための **“システムタイマー”** が用意された。

これは3.911マイクロ秒ごとに値が増える16ビットのカウンターで、下位バイトはI/OポートのE6H番地に、上位バイトはE7H番地に接続されている。ただし、16ビットの値を読もうとして、その途中でカウンターの値が変わると都合が悪いので、下位バイトと上位バイトの、どちらか一方のみを使うとよい。

リスト1.1は、Bレジスターの値×3.911マイクロ秒を待つプログラムの例だ。カウンターの下位バイトの代わりに、上位バイトの値を使うようにプログラムを書き替えれば、Bレジスターの値×1001.2マイクロ秒待つプログラムができる。

リスト 1.1 (TIMER.Z80)

```
        .Z80
COUNTLOW        EQU     0E6H    ; カウンター下位8ビット
COUNTHIGH       EQU     0E7H    ; カウンター上位8ビット
;
; Bレジスターの値*3.911uS待つ
; 誤差は-3.911uS..+0S
; 0を指定してはいけない
; 割り込みは禁止されていなければならない
; C、A、Fは破壊される
;
WAIT:
        IN      A,(COUNTLOW)    ; カウンターの現在値を得る
        LD      C,A             ; それを保存する
WAIT_LOOP:
        IN      A,(COUNTLOW)    ; カウンターの現在値を得る
        SUB     C               ; 経過時間を算出する
        CP      B               ; 指定された時間経過したか？
        JR      C,WAIT_LOOP     ; 経過していなければループする
        RET
```

1.1.6 MSX turbo RのI/Oポート

turbo Rの記者発表資料にはI/Oマップが含まれていなかったので、取材とハードウェアの解析によって得られた情報をMSX2+のI/Oマップに追加して、編集部が表1.1のI/Oマップを作った。“R”という注が付く項目が、turbo Rに新しく追加されたI/Oポートだ。

まず **“D/Aコンバーター”** というのは、PCMの録音再生を、BIOSをとおさずに操作するためのI/Oポート。あとで詳細を紹介しよう。**“ポーズキー制御”** は、ポーズキーによるプログラムの停止を禁止、許可するためのI/Oポート。ディスクの入

表 1.1: MSX turbo R の I/O マップ

番地	用途	注	
00H〜3FH	自作ハードウェア		
40H〜7BH	メーカーオプション		
7CH〜7DH	OPLL	+	B
80H〜87H	RS-232C	1	B、X
88H〜8BH	外部 VDP	2	XX
90H〜93H	プリンター	2	
98H〜9BH	VDP	+	
A0H〜A2H	PSG	1	
A4H〜A5H	D/A コンバーター	R	
A7H	ポーズキー制御	R	
A8H〜ABH	8255	1	B
ACH〜AFH	MSX-Engine	2	
B0H〜B3H	SONY の SRAM	1	XX
B4H〜B5H	時計	2	
B8H〜BBH	ライトペン	2	XX
BCH〜BFH	VHD 制御	2	XX
C0H〜C1H	MSX-Audio	2	XX
C8H〜CCH	MSX-Interface	2	XX
D0H〜D7H	フロッピーディスク	2	—、XX
D8H〜D9H	第 1 水準漢字 ROM	2	
DAH〜DBH	第 2 水準漢字 ROM	2	
DCH	漢字 ROM 拡張	R	—、X
E3H〜E5H	?	R	?
E6H〜E7H	システムタイマー	R	
F4H	リセットステータス	+	B
F5H	デバイスイネーブル	2	
F6H〜F7H	AV 制御	2	X
FCH〜FFH	メモリーマッパー	2	B

MSX マガジン編集部の調査による

- **1** **MSX_1 互換。**
- **2** **MSX_2 互換。**
- **+** **MSX_{2+}互換。**
- **R** **turbo R に新設。**
- **B** **かならず BIOS をとおして操作すること。**
- **—** **アプリケーションプログラムが操作してはいけない。**
- **X** **メーカーオプション機能。ただし今回調査した turbo R 試作機には、実装されていなかった。**
- **XX** **従来の仕様には含まれていたけれど、turbo R の仕様から削除された。**
- **?** **何かが接続されているが、仕様書には書かれていない。どうやら、ハードウェア検査用のレジスターがあるようだ。**

出力中にプログラムが中断され、ディスクが破壊されるような事態を防ぐために、用意されたようだ。

“漢字 ROM 拡張” は、24 ドットの漢字 ROM や、将来作られるかもしれない JIS 第 3 水準漢字 ROM に備えての、予約機能らしい。その下の “?” は、どの資料にも書かれていないのだけれど、I/O ポートを読み書きすると S1990 の内部で何らかのハードウェアが動作するようだ。turbo R のハードウェアを、工場で検査するための I/O ポートではないだろうか。そして “システムタイマー” は、既に説明したとおりのものだ。

なお、これは表には書いてないのだけれど、turbo R の速さに対応するためにも、PSG 、ジョイスティック、マウス、プリンター、キーボード、時計 (バッテリーバッ

クアップされたクロック IC) の操作には、BIOS を使うべきだ。

次に、MSX2+と共通の機能なのだけれど、補足説明しておきたいのが **“リセットステータス”** 。これは、ハードウェアのリセットと、メイン ROM の 0 番地へのジャンプによる再起動とを、区別するための I/O ポートだ。具体的には、メイン ROM の 17AH 番地をコールすると、このリセットステータスの値が A レジスターに読み出され、17DH 番地をコールすると、A レジスターの値がリセットステータスに書き込まれる。たとえば、

```
CALL  17AH
OR    80H
CALL  17DH
RST   0H
```

という手順で、リセットステータスのビット 7 を 1 にしてから 0 番地にジャンプすると、MSX を確実に再起動できるわけだ。

ところで、なぜ BIOS をとおさずにリセットステータスを使ってはいけないかというと、マシンによってリセットステータスのハードウェアの信号の論理が、逆になっているから。BIOS がその違いを補正しているというわけだ。

DOS2 が標準装備された turbo R で、ますます重要な存在になったのが **“メモリーマッパー”** 。やや複雑な手順で拡張 BIOS を使い、操作する必要があるものだ。

最後は余談になるけれど、表 1.1 には、かつて実用化または試作されたが、最近の MSX には搭載されていない機能もふくまれている。最近の MSX は当たり前のコンピューターになってしまい、カワリモノの周辺機器が少なすぎると筆者は思うのだが、どうだろうか。

1.1.7 速さを生かすための DRAM モード

メモリーにはそれぞれ、**“アクセスタイム”** と呼ばれる読み書きの最小時間間隔の制限がある。もしも CPU のスピードが速すぎた場合には、“ウェイト (待ち時間)” を入れて CPU の速さをメモリーに合わせる必要があるわけだ。このアクセスタイムは品種によって異なり、高速に使えるメモリーほど高価になる。また一般的に、ROM よりも RAM のアクセスタイムが短い。

さて R800 の速さを活かすには、プログラムが ROM より RAM に入っているほうがいい。そこで BIOS 、BASIC 、サブ ROM 、漢字ドライバーの各 ROM の内容を、DRAM(メイン RAM) に転送して使う、**“DRAM モード”** が用意された。

これは、メイン RAM の最後の 64 キロバイトをメモリーマッパーから切り離し、ROM の内容を転送してから書き込み禁止にし、CPU に接続するというもの。CPU

からは、普通の ROM が高速の ROM に差し替えられたように見える。BASIC で書かれたプログラムを実行させる場合など、BIOS と BASIC インタープリターが入った ROM がひんぱんに使われるので、DRAM モードの速さを生かせるわけだ。

しかし、マシン語のプログラム、とくに DOS のプログラムを実行させる場合は、ROM が使われる時間が比較的短い。そのため DRAM モードを使うより、余ったメモリーを RAM ディスクなどに活用するほうが有利かもしれない。

また、ROM カートリッジのプログラムも RAM に転送すると高速に動くけど、turbo R ではこれまで以上にディスク版のソフトウェアが主流になっていくだろう。

1.1.8 R800 の特徴はこれだ！

- Z80 とオブジェクトコンパチブル。だから Z80 用のソフトウェアも、CPU のタイミングに依存する部分を除いて動作する。

- CPU のクロック数は 7.16 メガヘルツ。しかも Z80 に比べて命令あたりのクロック数が大幅に減少しているため、Z80 に換算した場合は 29 メガヘルツに相当する (ただし、ノーウェイト時)。

- 16 ビット×16 ビット→32 ビットの精度を持つ乗算命令をサポート。これにより、演算処理速度の大幅な向上が可能になった。

- Z80 では未定義だった、IX / IY レジスターの、上位/下位 8 ビットごとのアクセスを、正式に保証した。

1.1.9 R800 のすべて

turbo R の CPU として採用された R800 は、従来の Z80 にソフトウェア上位互換の高速 CPU だ。つまり、CPU が速すぎて困らない限り、Z80 用に開発されたソフトウェアを、そのまま R800 で高速に実行することができる。

Z80 に追加された機能としては、16 ビットの乗算命令と、Z80 では “裏技” とされていた、IX/IY レジスターのバイトアクセスの命令。詳細については、本書の付録に R800 のインストラクション表を掲載するので、そちらを参考にしてほしい。

従来の MSX のクロック周波数は 3.58 メガヘルツで、turbo R のクロック周波数は 7.16 メガヘルツ。これだけ見ると、速度が 2 倍になっただけのように思えるけど、実際はそうじゃない。R800 ではひとつの命令を実行するのに必要なクロック数が減り、さらに RAM をアクセスするのに M1 サイクルのウェイトが発生しないので、プログラムの実行速度はさらに速くなる。従来の Z80 で、R800 と同じ処理速度を

達成するには、約29メガヘルツのクロック周波数になるというから、かなりのスピードアップがはかられたわけだ。

表 1.2: Z80 と R800 の動作速度を比較

命令		MSX2+ (単位μs)	turbo R (単位μs)	倍率
LD	r,s	1.40	0.14	x10.0
LD	r,(HL)	2.23	0.42	x 5.3
LD	r,(IX+n)	5.87	0.70	x 8.4
PUSH	qq	3.35	0.56	x 6.0
LDIR	(BC $\neq$ 0)	6.43	0.98	x 6.6
ADD	A,r	1.40	0.14	x10.0
INC	r	1.40	0.14	x10.0
ADD	HL,ss	3.35	0.14	x24.0
INC	ss	1.96	0.14	x14.0
JP		3.07	0.42	x 7.3
JR		3.63	0.42	x 8.7
DJNZ	(B $\neq$ 0)	3.91	0.42	x 9.3
CALL		5.03	0.84	x 6.0
RET		3.07	0.56	x 5.5
MULTU	A,r	—	1.96	—
MULTUW	HL,rr	—	5.03	—

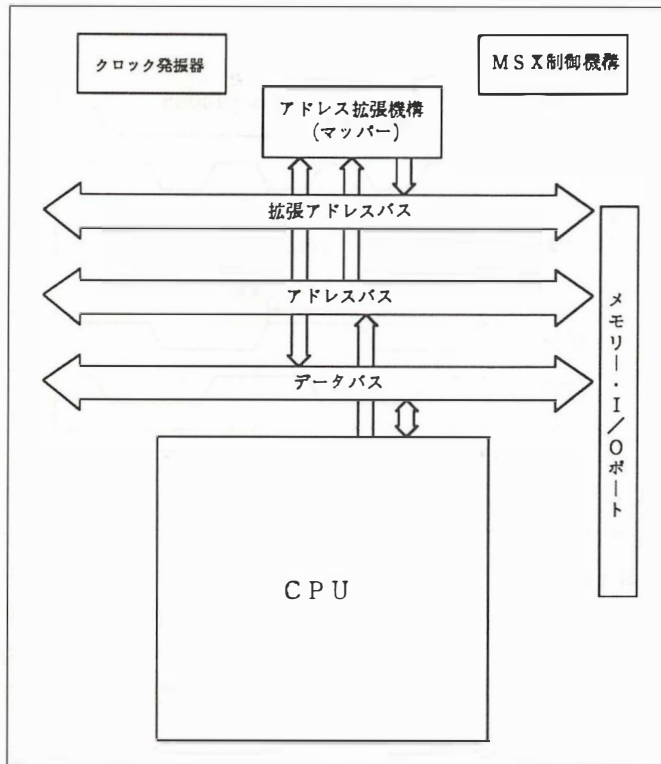
さて、命令の種類ごとに、Z80 と R800 の速さを比較してみた結果が表 1.2 。レジスター間のデータ転送 (LD 命令) と、加算の速さが 10 倍になることは、注目に値する。ただし、この表の値は、R800 がノーウェイトで動く場合の速さを測ったもの。実際にはウェイトによって速さが落ちる可能性もあるので、注意しよう。なお、ウェイトが発生する条件とその回避方法を、あとで詳しく説明する。

R800 の内部構造は、図 1.3 のようになっている。R800 では、外部データバスは 8 ビットなのだけど、CPU 内部のデータバスは 16 ビット。だから 16 ビットの加算命令などは、1 サイクルで処理されるわけだ。

このハードウェア構成を見てみると、R800 は 8 ビット CPU の Z80 よりも、外部データバスが 8 ビットの 16 ビット CPU 、たとえばインテル社の "8088" やモトローラ社の "MC68008" に近いといえそうだ。

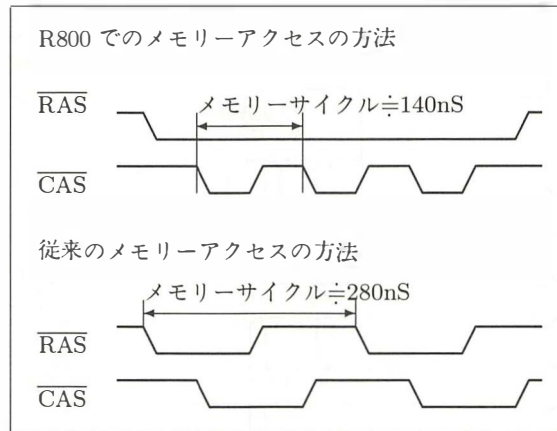
なお、図 1.3の上のほうに、"アドレス拡張機構 (マッパー)" というものがあるけれど、これは R800 を MSX 以外に使うために用意されたものらしい。turbo R で使う場合は、R800 ではなく S1990 に組み込まれたスロット制御機構と、メモリーマッパーがシステムを制御することになる。

それでは次に、**"DRAM のページアクセス"** を詳しく説明しよう。まず、これ

図 1.3: R800内部のブロック図

までの Z80 を使ったメモリーアクセスの方法を示したのが、図 1.4 の下側。アドレスの上位バイト (row address) を DRAM へ送り、RAS(row address strobe) 信号を LOW にし、アドレスの下位バイト (column address) を DRAM へと送ったあとで、CAS(column address strobe) 信号を LOW にする。これで、メモリーのアドレスが指定されるわけだ。

一方、R800 での DRAM のページアクセスを示したのが、図 1.4 の上側。アドレスの上位バイトと RAS 信号を固定したまま、アドレスの下位バイトと CAS 信号のみを変化させ、従来の方法の 2 倍の速さで、DRAM を使っている。このように、R800 ではアドレスの上位バイトが変わらずに、連続して DRAM が使われるとき、自動的にページアクセスが行なわれる。

さて、R800 に接続して使用するのが容易な DRAM の種類としては、256 キロビット (32 キロバイト)、1 メガビット (128 キロバイト)、4 メガビット (512 キロバイト) などがあげられる。メイン RAM 容量の最低値が、256 キロバイトと定められた

図 1.4: Z80 と R800 のメモリーアクセス方式の違い

turbo R でも、1 メガビットの DRAM がたった 2 個あれば、事足りてしまうわけだ。

1983 年に開発された最初の MSX では、16 キロビットの DRAM を 8 個も使い、それでもメイン RAM 容量は 16 キロバイトに過ぎなかった。そのことを思うと、わずか 2 個の DRAM で 256 キロバイトもの RAM 容量を達成する、現在の技術力はすごい。MSX の機能は大きくなったけれど、ハードウェアの大きさと消費電力は小さくなった。こうした、日本の最新の半導体技術の応用結果が、最近話題になっているノート型パソコンや、turbo R の登場といえるだろう。

1.2　MSX turbo R 活用法

1.2.1　R800 の速さを生かすプログラミング

確かに R800 は速いけれど、その速さを最大限に発揮するには、つまりウェイトを避けて R800 の能力を活用するには、プログラミングの工夫が必要になる。覚えておいてほしいのは、外部スロットのアクセスには 3 ウェイト、本体内蔵 ROM のアクセスには 2 ウェイト、本体内蔵 DRAM をページアクセスできなかったときには 1 ウェイトが発生することだ。

理想的には、本体内蔵 RAM の番地の上位バイトが同じような 256 バイトの範囲 (ページアクセス可能な範囲) にプログラムが置かれ、レジスターにデータが置かれるとよい。この場合には、データのためのメモリーアクセスが起こらず、CPU がプログラムを読むためのメモリーアクセスもページモードで行なわれるので、CPU にウェイトがかからない。すべてのプログラムを、こうして作ることは難しいけれど、もっとも速さを要求されるサブルーチンだけでも、この条件に近づけるといいだろう。

さて、ページアクセスの可否には、プログラム、データ、スタックの番地が関係する。たとえば、

```
PUSH    HL
```

命令の実行時間は、その命令が置かれている番地の上位バイトとスタックポインターの上位バイトが一致すれば 4 クロック。一致しなければ 5 クロックだ。ここまで考えながらプログラムを作る必要は少ないだろうけれど、**状況に応じて命令の実行時間が異なる**ことは重要なので、覚えておこう。

1.2.2　R800 を使う上での注意事項と問題点

Z80 では、ひとつの命令を実行するたびに DRAM をリフレッシュしていた。ところが R800 では、31 マイクロ秒ごとに 280 ナノ秒かけて、DRAM をリフレッシュする。注意してほしいのは、このリフレッシュに要する時間と、先ほど説明した DRAM のページアクセス可否の条件のため、R800 のプログラムの実行時間を正確に予測することができないことだ。

そこでプログラムの速さを調節するために、**“システムタイマー”** というものを使うことになる。あとで、このシステムタイマーの使い方と、CPU と VDP の間の速さの調整について説明するので、待っていてほしい。

また、これはどの新型 CPU でもいえることなのだけど、R800 の問題点として考えられるのは、開発機材が不足していること。とくに、ソフトウェアを開発すると

きに威力を発揮する“ICE(インサーキットエミュレーター)”を、デバッグに使えないことが不便だ。

そのため、turbo R 用のソフトウェアを作るためには、まず従来の MSX と Z80 用の ICE を使って徹底的にデバッグし、確実に動くはずのプログラムを turbo R 用に直す方法がいいだろう。Z80 兼用のプログラムを作って動作を確認してから、掛け算を使う部分のみを R800 用に書き替えるわけだ。このとき、サブルーチンごとにわけて、動作をチェックするのもいい。そして、最後に全体を組み立てて動かなければ……ソースリストを見て考えるしかない。

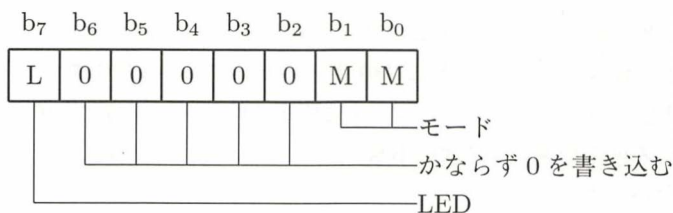
1.2.3 追加された BIOS とその機能説明

turbo R の新しいハードウェア機能を制御するために、CPU の切り替えと、PCM の録音再生のための BIOS が追加された。

ここでは、BIOS の名称 (ラベル)、エントリーアドレス (番地)、そして機能と各レジスターの順番で説明する。BIOS の機能を書き表わすための記号は、以下のとおりだ。まず、E とは BIOS を呼び出す前に値を設定すべきレジスター。R は BIOS が値を返すレジスターで、M は BIOS が無意味な値を書き込む、つまり元の内容が壊されるレジスターを表わす。また IYH とは、IY レジスターの上位バイトを表わし、下位バイトの内容は無視される。

CHGCPU 0180H 番地

機能　CPU を切り替える。

E　A レジスターのビット 1 と 0 で、次のようにモードを設定する。このうち“R800 DRAM”というのは、BIOS の ROM の内容を DRAM に転送して使うモードのことだ。

モード	
00	Z80
01	R800 ROM
10	R800 DRAM

また、Aレジスターのビット7が1ならば、どちらのCPUが動いているかを表わすLEDが変化する。逆にAレジスターのビット7が0なら、CPUが切り替えられるが、LEDは変化しない。

R　なし

M　AF

注　CPUを切り替える前のレジスターの内容は、AFとRを除いて、切り替え後のCPUにそのまま引き継がれる。また、切り替えたあとは割り込みが許可される。なおCPU切り替えの注意事項については、あとで詳しく説明する。

GETCPU　0183H番地

機能　動作中のCPUを調べる。

E　なし

R　動作中のCPUに応じて、Aレジスターに次のような値が返される。

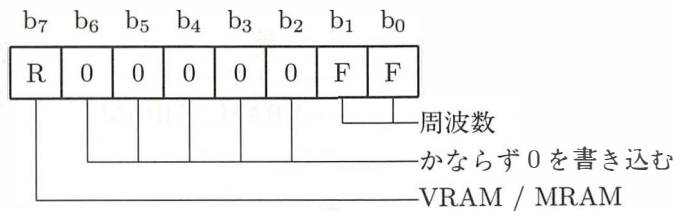
0	Z80
1	R800 ROM
2	R800 DRAM

M　F

注　あとで説明する方法でハードウェアがturbo Rであることを確かめてから、このBIOSを呼び出す必要がある。

PCMPLY　0186H番地

機能　PCMの音を再生する。

E　A

EHL (データの番地)

DBC (データの長さ)

Aレジスターのビット7が1ならばビデオRAMに、0ならばメインRAMにPCMの音源データが置かれる。なおビデオRAMにデータがある場合にのみ、DレジスターとEレジスターの値が意味を持つ。

Aレジスターのビット1とビット0で、サンプリング周波数を設定する。ただし15.75キロヘルツは、turbo RがR800のDRAMモードで動いている場合だけ指定可能だ。

00	15.75キロヘルツ
01	7.875キロヘルツ
10	5.25キロヘルツ
11	3.9375キロヘルツ

R　キャリーフラグ
0　正常終了
1　異常終了

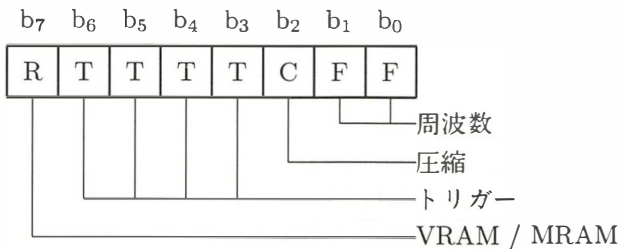
A (異常の原因)
1　周波数指定誤り
2　STOPキーによる中断

EHL (中断番地)

M　すべて

PCMREC　0189H番地

機能　PCMの音を記録する。

E　A

EHL (データの番地)

DBC (データの長さ)

Aレジスターのビット7、1、0の設定方法は、PCMPLYで説明したものと同じ。Aレジスターのビット6からビット3は“トリガーレベル”と

いい、録音をはじめるきっかけとなる音の大きさを指定する。この値が0ならば、ただちに録音が開始される。

また、A レジスターのビット 2 が 1 ならば、録音データが圧縮される。0ならば圧縮されない。

R　キャリーフラグ
- 0　正常終了
- 1　異常終了

A (異常の原因)
- 1　周波数指定誤り
- 2　STOP キーによる中断

EHL (中断番地)

M　すべて

1.2.4　変更および削除された BIOS について

turbo R で変更または削除された BIOS は、表 1.3 に示したとおり。それぞれについて、簡単に説明していく。

まず、turbo R ではカセットテープインターフェースがなくなったので、従来の BIOS にあった "TAPION"、"TAPIN"、"TAPIOF"、"TAPOON"、"TAPOUT"、"TAPOOF" をコールすると、キャリーフラグがセットされ、エラーとしてリターンする。また、"STMOTR" もなくなり、コールしても何もしないでリターンする。

また、メイン ROM の容量を変えずに新しい機能を追加するために、パドルとライトペンの BIOS が削除された。BIOS の "GTPDL" をコールすると、A レジスターにかならず 0 が入ってリターンする。同様に、"GTPAD" または "NEWPAD" で、A レジスターにライトペンを指定する 8~11 の値を入れてコールしても、A レジスターにはかならず 0 が入ってリターンする。

変更された BIOS としては、使用中の MSX のバージョンを知るための、"ROM version ID"。これはメイン ROM の 002DH 番地の内容でわかり、turbo R の場合は 03H に変更された。turbo R 用にプログラムを開発するなら、まずこの番地の値が 03H 以上であることを確かめ、そうでなければ、MSX2 用のプログラムとして動作させるか、あるいはエラーメッセージを表示して中断させるようにしよう。

なお、002DH 番地の内容が 03H の場合のみに動くようなプログラムは、将来 MSX がバージョンアップしたときに動かなくなってしまうので、かならず 03H 以上ならば動くように作る必要がある。一般的に、ハードウェアや OS のバージョンについ

表 1.3: MSX turbo R で変更のあった BIOS と BASIC の一覧

追加された BIOS エントリー	
CHGCPU	0180H
GETCPU	0183H
PCMPLY	0186H
PCMREC	0189H

変更された BIOS エントリー	
ROM version ID	002DH

削除された BIOS エントリー	
GTPDL	00DEH
TAPION	00E1H
TAPIN	00E4H
TAPIOF	00E7H
TAPOON	00EAH
TAPOUT	00EDH
TAPOOF	00F0H
STMOTR	00F3H
GTPAD	00DBH
NEWPAD	SUB 01ADH

追加されたステートメント
CALL PCMREC
CALL PCMPLAY
CALL PAUSE

変更されたステートメント
COPY

削除されたステートメント
CLOAD
CSAVE
MOTOR

ては、自分が必要とするバージョン番号以上の値を得られれば、ソフトウェアが動作するようにプログラムしておこう。

これは過去において実際にあったことなのだけれど、MSX のバージョンのチェックを誤ったために、MSX2+では動かない MSX2 用プログラムや、学習機能付きの MSX–JE と組み合わせると動かないアプリケーションなどが、できてしまう。それを避ける意味でも、“03H 以上なら動くようにする” ということを、忘れないでほしい。

また、BIOS と同様に、turbo R になっての BASIC の機能にも追加や変更、削除があった。それらについては、表 1.3 や、マシン付属の BASIC マニュアルを参照してほしい。

1.2.5 アプリケーション開発に関する注意点

MSX turbo R では、R800 は常にノーウェイトで動作しているわけではない。外部スロットをアクセスするときに 3 ウェイト、内部 ROM をアクセスするのに 2 ウェイト、そして内部 DRAM がページブレークを起こしたときに 1 ウェイトかかるのだ。そこで、プログラムの高速化をはかるには、こうしたウェイトをできる限り減らすことを考えながら、作業しなくてはいけない。そのための注意点を 3 つほどまとめてみたので、覚えておこう。

まずひとつ目は、プログラム自体を RAM に転送してから実行させること。フロッピーで供給されるソフトウェアは、必然的に RAM で動作するので問題ないのだけれど、注意したいのはスロット上に ROM カートリッジで供給されるプログラム。必要な部分だけを RAM に転送してから実行させることで、かなりの高速化が可能になる。

ページブレークを起こさないようにコーディングすることも大切だ。R800 では、DRAM のページアクセスをサポートする専用のバスを持っているので、この機能を最大限に活用しよう。具体的には、アドレスの下位 8 ビットだけが変化するような連続したメモリー、つまり??00H~??FFH までの 256 バイトの範囲で、メモリーアクセスが行なわれるようにプログラムするのが効果的だ。

ちなみに、ページブレークを起こした状態というのは、この範囲を越えてメモリーアクセスが行なわれた場合、つまりアドレスの上位 8 ビットが変化した場合のことを呼んでいる。

前にもちらっと書いたのだけれど、turbo R では MSX2+などとは違い、プログラムのコーディング段階で命令の実行時間が正確にわかるわけではない。その理由としてあげられるのが、いつ発生するか予測のつかない DRAM のページブレークと、Z80 などとは違って、命令の実行とは非同期に行なわれる DRAM のリフレッシュがあるからだ。

また、turbo R と MSX2+のどちらでも動作するようなプログラムを作るのに、ソフトウェアループによってタイミングをとることは勧められない。そこで turbo R には、3.911 マイクロ秒ごとにカウントアップするシステムタイマーが、新たに搭載された。これからは、このシステムタイマーを利用して、タイミングをとるようにしよう。

1.2.6 CPU を切り替えるプログラムの例

リスト 1.2 は、CPU を切り替える "CHGCPU.COM" のソースリストだ。turbo R で DOS2 が動いているときに、

```
CHGCPU 0
```

で Z80 モードが、

```
CHGCPU 1
```

で R800 の ROM モードが、

```
CHGCPU 2
```

で R800 の DRAM モードが、それぞれ選択される。プログラムの内容を解説すると、DOS のワークエリア (正確には default FCB area) の 5DH 番地からコマンドの第 1 パラメーターの先頭の文字を得て、それに応じて A レジスターの値を設定。そして、メイン ROM の 180H 番地の BIOS、“CHGCPU” を呼び出すというものだ。

また、プログラムにより実用性を持たせるため、DOS のバージョン番号をチェックする処理も加えてある。具体的には、まずメイン ROM の 2DH 番地の内容が 03H 以上である、つまり turbo R であることを確かめ、DOS のシステムコールの 6FH を使って、DOS カーネルのバージョン番号が 2 以上であることを確かめている。

リスト 1.2 (CHGCPU.Z80)

```
        .Z80
RDSLT   EQU     0000CH          ; inter slot read
CALSLT  EQU     0001CH          ; inter slot call
EXPTBL  EQU     0FCC1H          ; slot # of main ROM
;
        ld      a,(EXPTBL)      ;
        ld      hl,2dh          ; address to read
        call    RDSLT           ; read version
        cp      3
        jr      nc,TURBOR
        ld      de,MSG_NOTR
        ld      c,9             ; _STROUT
        call    5
        rst     0               ; return to DOS
TURBOR:
        ld      c,6fh           ; _DOSVER
        call    5               ;
        ld      a,b             ; version of DOS kernel
        cp      2
        jr      c,NOTDOS2       ;
        ld      a,d             ; version of MSXDOS.SYS
        cp      2
        jr      c,NOTDOS2

        ld      a,(005ch+1)     ; command parameter
        sub     '0'             ; 0:Z80, 1:R800ROM, 2:R800RAM
        ret     c               ; abort if parameter < '0'
        cp      3
        ret     nc              ; abort if '3' <= parameter
        or      80h               set change-LED flag
        ld      ix,180h         ; address of CHGCPU
        ld      iy,(EXPTBL-1)   ; slot of main ROM
        call    CALSLT          ; inter-slot call
        rst     0               ; return to DOS
;
NOTDOS2:
        ld      de,MSG_NOTDOS2
        ld      c,9             ; _STROUT
```

```
        call    5
        rst     0               ; return to DOS
;
MSG_NOTR:
        DB      'not MSX turbo R', 0dh, 0ah, '$'
MSG NOTDOS2:
        DB      'not MSX-DOS 2', 0dh, 0ah, '$'
        END
```

同様に、次のリスト 1.3 は、MSX2 用のプログラムを R800 モードでだまして動かす、“GAMEBOOT.COM” のソースリストだ。このプログラムは、DOS2 が起動され R800 が選択されている状態で、ほかのディスクに入っているプログラムを起動するためのもの。つまり、DOS2 のシステムが含まれていないプログラム (ゲームなど) を、強引に R800 モードで動かすためのものだ。

簡単にプログラムを解説していくと、まず画面にメッセージを表示して、ディスクが交換されるのを待つ。次に、交換されたディスクのブートセクターを読み込んで、それを実行させる。そのときの環境は、普通の方法でブートセクターが 2 回目にコールされるときと同じで、ページ 1 は DOS の ROM 、そのほかのページは RAM 、キャリーフラグはセットされている。

さらに、エラー処理プログラムへのポインターの、ポインターを記憶するための DOS のワークエリア (F323H 番地) を HL レジスターに、またページ 1 を RAM から DOS の ROM へ切り替えるプログラムの番地 (F368H 番地) を DE レジスターに、それぞれ設定している。

リスト 1.3 (GAMEBOOT.Z80)

```
        .z80

_conin          equ     01h
_strout         equ     09h
_setdta         equ     1ah
_rdabs          equ     2fh

dos             equ     0005h
enaslt          equ     0024h

notfirst        equ     0f340h
master          equ     0f348h

        ld      sp,(6)
        ld      de,prompt       ; print prompt message
        ld      c,_strout
        call    dos
        ld      c,_conin        ; wait for key in
        call    dos
        ld      de,0c000h       ; read boot sector at 0c000h
        ld      c,_setdta
        call    dos
        ld      de,0            ; logical sector 0
        ld      l,0             ; drive A:
        ld      h,1             ; read 1 sector
        ld      c,_rdabs
        call    dos
        ld      h,40h
        ld      a,(master)
        call    enaslt
        ld      hl,0f323h
        ld      de,0f368h
        xor     a
        ld      (notfirst),a
        scf
        jp      0c01eh
prompt:
        db      'Insert game disk in drive A:,',0dh,0ah
        db      'and press any key $'

        end
```

1.3 PCM 限界ギリギリ活用法

turbo R に加えられた新しい機能が PCM 。せっかく用意された機能だから、その性能をギリギリまで引き出したいと思うのが人情だ。そこで、BASIC からマシン語、水平走査線割り込みを利用した特殊な使い方まで、PCM の活用法を紹介する。

1.3.1 基礎編…… BASIC での使い方

まずは BASIC を使った基本的なものから紹介しよう。

そもそも PCM は、マイクなどから入力した音声をデジタルに変換してメモリーに記憶させ、任意にそれを再生させるものだ。

turbo R の場合、PCM データを記憶するのは、メイン RAM かビデオ RAM 。サンプリングレートは、15.75 キロヘルツ、7.875 キロヘルツ、5.25 キロヘルツ、3.9375 キロヘルツの 4 種類から選択することになる。この値が大きいほど、より質の高いサンプリングができるというわけだ。

BASIC から PCM を使う場合は、ふたつの命令を覚えておけばいい。使い方はあとにまとめておいたので、参考にしてね。基本的には、これらの命令を実行するだけで、PCM の録音や再生は可能になる。ただし、データを記憶する開始番地と終了番地の設定には、十分に注意する必要があるぞ。

まず最初に、BASIC の “CLEAR” 命令で PCM データ用のメモリー領域を確保しておかないと、間違いなく暴走してしまう。たとえば C000H~D000H 番地までを PCM データ用に使うときは、

```
CLEAR 200,&HC000
```

のようにしておこう。とりあえず、リスト 1.4 に簡単なサンプルプログラムを載せておいたので、これを入力して遊んでみるといいだろう。

もちろん、ビデオ RAM を PCM データ用に使う場合は、任意のどの番地にもデータを置くことができので、開始番地や終了番地を気にする必要はない。それにビデオ RAM の場合は、PCM データを目で確認することができる。最初に

```
SCREEN 8
```

のように、スクリーンモードを設定してから PCM 録音すれば、画面にデータがズラズラっと表示されて、おもしろいかもしれない。

基本的な PCM の録音、再生の方法は、以上のことを注意すれば大丈夫。さらに再生サンプリングレートを変化させれば、4 段階のスピードで再生することもできる。ただ、問題となるのは、PCM を再生しているとき。turbo R がそれにかかりっ

きりになってしまうので、PCMを再生しながら何かをするなんてことは、残念ながらできないのだ。

リスト 1.4 (PCM1.BAS)

```
10 CLEAR100,&H9000
20 PRINT "イマカラ ロクオン シマス。";
30 A$=INPUT$(1):PRINT
40 _PCMREC  (@&H9000,&HCFFF,0)
50 PRINT "サイセイ シマス。";
60 A$=INPUT$(1):PRINT
70 _PCMPLAY (@&H9000,&HCFFF,0)
80 GOTO 20
```

1.3.2 PCM関係のBASIC命令

CALL PCMREC

書式

- メインRAMまたはビデオRAMへの録音。
  CALL PCMREC(@開始番地, 終了番地, サンプリングレート [, [トリガーレベル], 圧縮スイッチ][, S])

- 配列変数への録音。
  CALL PCMREC(配列変数名, [長さ], サンプリングレート [, [トリガーレベル], 圧縮スイッチ])

サンプリングレートの設定	
指定値	サンプリングレート
0	15.7500KHz
1	7.8750KHz
2	5.2500KHz
3	3.9375KHz

トリガーレベルでは、録音が開始されるときの入力レベルを設定する。値は0~127まで。この値以上の入力レベルになると録音が開始され、0または省略した場合には、すぐに録音がはじまる。圧縮スイッチの設定は、1で無音部分を圧縮し、0または省略すると圧縮しない。

CALL PCMPLAY

書式

- メインRAMまたはビデオRAMからの再生
  CALL PCMPLAY(@開始番地, 終了番地, サンプリングレート [, S])

- 配列変数からの再生
  CALL PCPLAY(配列変数名, [長さ], サンプリングレート)

PCMREC 、PCMPLAY ともに、高速モードでないときは、一時的に高速モードにしてから実行し、終了するともとの状態に戻ってくる。また、R800 の ROM モードで 15.75KHz が指定された場合は、エラーになる。
録音、再生中に[STOP]キーが押されると、プログラムの実行は中断される。PCM データの形式は、1~255 までが通常のデータで、0 は特殊なもの。あとに続く 1 バイトで指定された回数分だけ、0 レベル (127) を出力する。

1.3.3 BEEP 音を PCM で鳴らすのだ！

BASIC のプログラム実行しているときに、[CTRL] と [STOP] キーを同時に押してプログラムを中断させると、“ピッ” と BEEP 音が鳴るのは知ってるよね。“LIST” 命令でリストを表示させ、[CTRL] + [STOP] で止めたときも、同じように “ピッ” と音がする。BASIC の “SET BEEP” 命令を使えば、音を変えることもできるけど、4 種類用意されているどの音も、いまひとつインパクトに欠けるのだ。

そこで、この BEEP 音を PCM で鳴らすとどうなるか。変なセリフを設定しておくと、ことあるごとに MSX がしゃべるので、けっこううるさくて楽しいかもしれない。

というわけで、リスト 1.5 の掲載したプログラムを実行すると、BEEP 音を PCM で鳴らせるようになる。もちろん turbo R 専用だ。それほど長いものでもないので、頑張って入力してほしい。

なお、このプログラムは、メイン RAM のページ 1(4000H~60FFH 番地まで) に置かれるので、プログラムを実行したあと、“CLEAR” 命令でユーザーエリアの上限を B000H 番地以上にしてもかまわない。ただし、メモリーディスク関係は使えないので、うっかり “CALL MEMINI” なんてやらないように。それから、BASIC の “BEEP” 命令を使うときは、

```
PRINT CHR$(7)
```

を代わりに使わないと、BEEP 音が PCM にならない。注意しよう。

プログラムの使い方を説明する。

1 PCM BEEP セット

以降 BEEP 音が PCM になる。1 回実行しておけば、電源を切るまで設定は有効。また、CLEAR 文の設定を変更してもかまわない。

2 PCM BEEP リセット

BEEP 音を、もとの状態に戻す。“CALL SYSTEM” で DOS や DOS2 にするときは、かならずこのコマンドを実行すること。

3 PCM データ再生

現在設定されている PCM データを再生する。確認用に使おう。

4 PCM データ録音

PCM データを 15.75 キロヘルツで録音する。録音時は B000H~CFFFH 番地までのメモリーを使用。

5 PCM データ LOAD

拡張子が“.PCM”の、BSAVE 形式でセーブされた PCM データを読み出す。

6 PCM データ SAVE

“PCM データ録音”で録音した PCM データを、ディスクに記録する。

0 END

プログラムを終了する。もちろん、CTRL + STOP でもかまわない。

なお、簡単なメッセージが画面に表示されるので、参考にしよう。

リスト 1.5 (PCM2.BAS)

```
10 SCREEN0:WIDTH40:DEFINT A-Z
20 CLEAR100,&HB000
30 DEFUSR=&HD800:DEFUSR1=&HD806:DEFUSR2=&HD803
40 FOR I=&HD800 TO &HD87F
50 READ A$:POKE I,VAL("&H"+A$):NEXT
100 PRINT
110 PRINT"1) PCM BEEP セット  "
120 PRINT"2) PCM BEEP リセット"
130 PRINT"3) PCM DATA サイセイ"
140 PRINT"4) PCM DATA ロクオン"
150 PRINT"5) PCM DATA LOAD"
160 PRINT"6) PCM DATA SAVE"
170 PRINT"0) END"
180 PRINT" ....HIT 0-6 KEY=";
190 A$=INPUT$(1):I=ASC(A$)-ASC("0")+1
200 IF I>0 AND I<8 THEN ELSE190
210 ON I GOTO 230,240,310,220,340,390,430
220 PRINTCHR$(7);:GOTO 190
230 GOSUB 470:END
240 GOSUB 470:I=USR1(0):I=USR(0)
250 PRINT"PCM BEEP ヲ ツカウコトガ デキマス。"
260 PRINT"DOS マタハ DOS2 ヲ ツカウトキハ カナラズ PCM BEEP ヲ リ
セット シナオシテ クダサイ。"
270 PRINT"PCM BEEP ノ データ ハ ページ1(4100H カラ 60FFH) ニ アリ
マス。"
280 PRINT"CLEAR メイレイ デ B000H イジョウ ニ セッテイ シテモ カ
マイマセンガ、CALL MEMINI ナドノ メモリーディスク カンケイ ノ メ
イレイ ハ ツカウ コトガ デキマセン。"
```

```
290 PRINT"ナオ、BEEP メイレイ ヲ シヨウ スルトキ ハ PRINT CHR$(7
) ヲ ツカッテ クダサイ。"
300 END
310 GOSUB 470:POKE &HFDA4,&HC9
320 PRINT"PCM BEEP ヲ リセット シマシタ。"
330 GOTO 100
340 GOSUB 470
350 PRINT"PCM ロクオン ヲ ハジメマス。 (HIT ANY KEY!)";
360 A$=INPUT$(1):PRINT:_PCMREC(@&HB000,&HCFFF,0):I=USR(0)
370 PRINT"ロクオン シュウリョウ デス。"
380 GOTO 100
390 GOSUB 470
400 PRINT"PCM データ LOAD"
410 INPUT" FILE NAME(8 モジ)=";A$
420 BLOAD A$+".PCM":I=USR(0):GOTO 100
430 GOSUB 470
440 PRINT"PCM データ SAVE"
450 INPUT" FILE NAME(8 モジ)=";A$
460 I=USR2(0):BSAVE A$+".PCM",&HB000,&HCFFF:GOTO 100
470 PRINT CHR$(I+47):PRINT:PRINT:RETURN
480 DATA C3,4D,D8,C3,5F,D8,CD,6D
490 DATA D8,3A,42,F3,32,2A,D8,21
500 DATA 2E,D8,11,00,40,01,00,01
510 DATA ED,B0,CD,76,D8,21,29,D8
520 DATA 11,A4,FD,01,05,00,ED,B0
530 DATA C9,F7,00,00,40,C9,FE,07
540 DATA C0,01,00,20,21,00,41,3E
550 DATA 03,D3,A5,F3,DB,A4,D6,01
560 DATA 38,FA,7E,D3,A4,23,0B,79
570 DATA B0,20,F1,FB,C9,CD,6D,D8
580 DATA 21,00,B0,11,00,41,01,00
```

1.3.4 上級編……マシン語で PCM を！

マシン語で PCM を使う場合、手っとり早いのが BIOS を使う方法。サンプリングレートや、トリガーレベルなどの設定は、BASIC のものとほぼ同じなので問題はないだろう。

ここでは上級編ということなので、この BIOS を使わずに、PCM を録音したり再生したりできるプログラムをふたつ紹介する。

BIOS を使う場合、サンプリングレートは 15.75 キロヘルツ、7.875 キロヘルツ、5.25 キロヘルツ、3.9375 キロヘルツの 4 種類しか選べない。63.5 マイクロ秒ごとに値が変わるカウンターを使っているため、4 種類以上のサンプリングレートを設定できないのだ。このカウンターを、3.911 マイクロ秒ごとに値が変更される、システムタイマーで肩代りしたのがここで紹介するプログラムだ。

まずは、録音プログラムの使い方から説明しよう。HL レジスターには PCM データを格納するメモリーの先頭アドレスを、BC レジスターには録音するデータの大きさを、それぞれ設定しておく。そして E レジスターには、システムタイマーで何

カウント分のウェイトを入れるのか、を設定する。16で、だいたい15.75キロヘルツに相当するかな。

再生プログラムのほうも同じ。HLレジスターに再生するPCMデータの先頭アドレス、BCレジスターにはデータの大きさ、そしてEレジスターにウェイトのカウント数を設定すればいい。

PCM録音プログラムのリストの途中に、

```
0EDH,70H
```

というヘンなものがあるけど、これは

```
IN (HL),(C)
```

という命令のこと。Cレジスターのポートから値を読み、それをフラグだけに反映させる、R800独自の命令だ。

プログラムの原理はともあれ、とにかく使ってみよう。Eレジスターの値を変えることで、音がいろいろに変化して楽しめるはずだ。

リスト 1.6 (PCMREC.MAC)

```
PMDAC   EQU 0A4H
PMCNT   EQU 0A4H
PMCNTL  EQU 0A5H
PMSTAT  EQU 0A5H
SYSTML  EQU 0E6H        ; system timer port

REC:
        LD     A,00001100B
        OUT    (PMCNTL),A       ; A/D MODE
        DI
        XOR    A
        OUT    (SYSTML),A       ; reset timer

REC1:
        IN     A,(SYSTML)
        CP     E
        JR     C,REC1           ; wait
        XOR    A
        OUT    (SYSTML),A       ; reset timer

        PUSH   BC
        LD     A,00011100B
        OUT    (PMCNTL),A       ; DATA HOLD
        LD     A,80H
        LD     C,PMSTAT

        OUT    (PMDAC),A        ; BIT CONVERT
```

```
        DEFB  0EDH,70H              ; IN (HL),(C)
        JP    M,RECAD0
        AND   01111111B
RECAD0:
        OR    01000000B

        OUT   (PMDAC),A
        DEFB  0EDH,70H
        JP    M,RECAD1
        AND   10111111B
RECAD1:
        OR    00100000B

        OUT   (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD2
        AND  11011111B
RECAD2:
        OR   00010000B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD3
        AND  11101111B
RECAD3:
        OR   00001000B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD4
        AND  11110111B
RECAD4:
        OR   00000100B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD5
        AND  11111011B
RECAD5:
        OR   00000010B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD6
        AND  11111101B
RECAD6:
        OR   00000001B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD7
        AND  11111110B
RECAD7:
        OR   00000000B
```

```
        LD   (HL),A
        LD   A,00001100B
        OUT  (PMCNTL),A

        POP  BC
        INC  HL
        DEC  BC
        LD   A,C
        OR   B                  ; end of data ?
        JR   NZ,REC1            ; next data

        LD   A,00000011B
        OUT  (PMCNTL),A   ;D/A MODE
        EI
        RET

        END
```

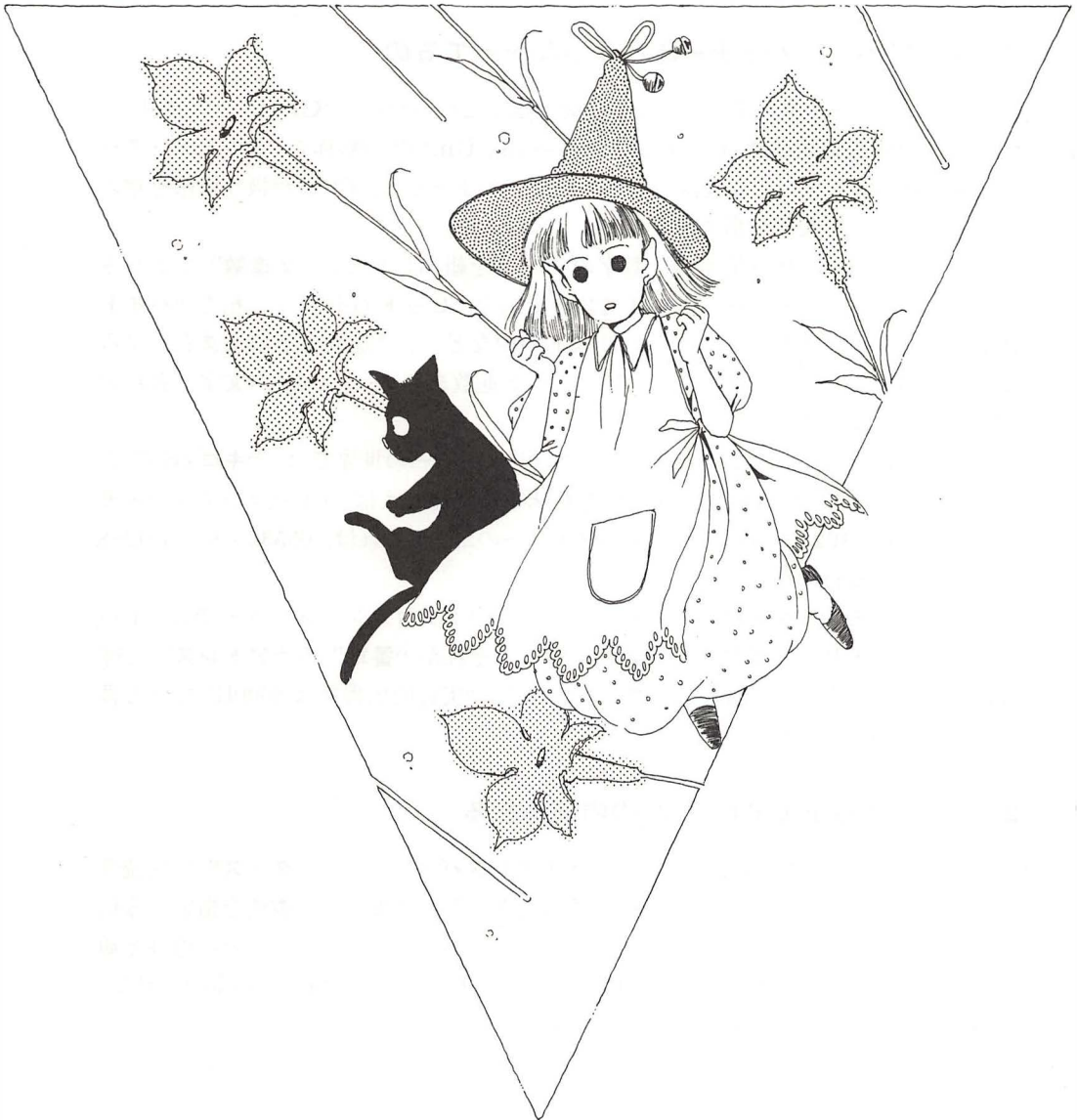
リスト 1.7 (PCMPLAY.MAC)

```
PMDAC   EQU 0A4H
PMCNT   EQU 0A4H
PMCNTL  EQU 0A5H
PMSTAT  EQU 0A5H
SYSTML  EQU 0E6H         ; system timier port

PLAY:
        LD   A,00000011B
        OUT  (PMCNTL),A         ; D/A MODE
        DI
        XOR  A
        OUT  (SYSTML),A         ; reset timer

PLAY1:
        IN   A,(SYSTML)
        CP   E
        JR   C,PLAY1            ; wait
        XOR  A
        OUT  (SYSTML),A         ; reset timer

        LD   A,(HL)
        OUT  (PMDAC),A          ; play 1 byte
        INC  HL
        DEC  BC
        LD   A,C
        OR   B                  ; end of data ?
        JR   NZ,PLAY1           ; next data
        EI
        RET

        END
```

表 1.4: PCM 用の I/O ポート

番地	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0A5H Write	0	0	0	SMPL	SEL	FILT	MUTE	ADDA
0A5H Read	COMP	0	0	SMPL	SEL	FILT	MUTE	BUFF
0A4H Write	DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0
0A4H Read	0	0	0	0	0	0	CT1	CT0

- **ADDA (BUFF)**:バッファーモード
  D/A コンバーターの出力を指定する。D/A 時は 0(ダブルバッファー)、A/D 時は 1(シングルバッファー) にしよう。なお、リセット時はダブルバッファーの状態になっている。
- **MUTE**:ミューティング制御
  システム全体の音声出力をオンにしたり、オフにしたりする。
  0:音声出力オフ (リセット時)
  1:音声出力オン
- **FILT**:サンプルホールド回路入力信号の選択
  A/D 時にサンプルホールド回路に入力する信号を、フィルターの出力信号にするか、基準信号にするかを選択する。0 で基準信号、1 でフィルター出力信号音になる。リセット時は 0 。
- **SEL**:フィルター入力信号の選択
  ローパスフィルターに入力する信号を、D/A コンバーターの出力信号にするか、マイクアンプの出力信号にするかを選択する。0 で D/A コンバーター出力信号、1 でマイクアンプ出力信号音。
- **SMPL**:サンプルホールド信号
  入力信号をサンプルするか、ホールドするかを選択する。
  0:サンプル (リセット時)
  1:ホールド
- **COMP**: コンパレーターの出力信号
  サンプルホールドの出力信号と D/A コンバーターの出力信号とを比較する。
  0:D/A 出力>サンプルホールド出力
  1:D/A 出力<サンプルホールド出力
- **DA7~DA0**: D/A 出力データ
  PCM データを再生するときに、用意されたデータをここに出力することで、PCM 音を再生することができる。データの形式はアブソリュートバイナリーで、127 が 0 レベルに相当する。
- **CT1 、CT0**:カウンターデータ
  63.5 マイクロ秒ごとにカウントアップされる。D/A 時にはカウントアップに同期し、0A4H 番地に書かれたデータが繰り返し出力される。また 0A4H にデータを書き込むとカウンターはクリアーされる。

第2章 SLOT

この章は、MSXマガジン1989年2月号、1989年3月号の“MSX2+テクニカル探検隊”と、1990年11月号の“テクニカル・アナリシス”の記事を再編集したものである。

2.1 スロットって何だ

MSXにカートリッジをセットするための穴は“カートリッジスロット”。でもスロットが意味するのは、MSXのメモリーを管理する機能でもある。この章では、もっとも重要で難解なスロットを説明するぞ。

2.1.1 CPUとメモリーはどうつながってるの

コンピューターを構成するもっとも重要な部品といったら、**“CPU”**とメモリー。CPUとは**“中央処理装置(Central Prosessing Unit)”**の略称で、コンピューター全体を管理し計算を行なう装置のこと。一方メモリーとは、CPUが扱う情報を覚えるメモ帳のような装置を指す。

コンピューターが扱う情報は、数字の0と1を組み合わせた**“2進数”**で表わされることは知ってのとおり。この2進数の1桁を**“ビット(bit)”**、8桁を**“バイト(byte)”**という。また、プログラムリストの中などで、2進数をそのまま表記すると桁数が多くなってしまうので、4ビットの2進数を0~9とA~Fの文字で表わす**“16進数”**もよく使われる。

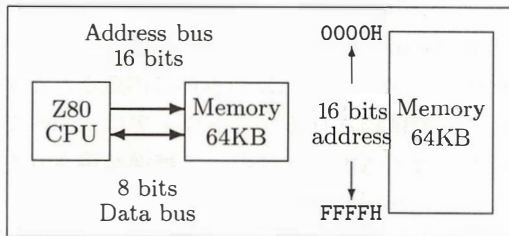
これは間違えないでほしいのだけど、コンピューターの世界では、**“キロ(K)”**という単位が、1000倍ではなく1024倍を意味する。たとえば、64キロバイトのメモリーとは、$64 \times 1024 = 65536$バイトのメモリーのこと。これは、$65536 \times 8 = 524288$ビットでもあるわけだ。

これらのメモリーを管理するために、多くのマイクロコンピューターでは、1バイトごとにメモリーに番号が付けられている。それが**“番地”**や**“アドレス”**と呼ばれるもの。よくマシン語のプログラムなどで、“実行開始番地は8000H”などと書かれているのがそうだ。

2.1.2 8ビットCPU Z80の内部を探る

CPUとメモリーは、図2.1のように**“アドレスバス”**と**“データバス”**で接続されている。アドレスバスとは、CPUが読み書きしたいメモリーの番地を指定する信号を、CPUからメモリーへ送るための電線。データバスとは、メモリーの内容を通信するための電線のこと。前者がCPUからメモリーへの一方通行であるのに対し、後者は双方向になっている点に注意しよう。

図 2.1: Z80 CPU のメモリー

Address bus
16 bits
Z80
CPU
Memory
64KB
8 bits
Data bus
0000H
16 bits
address
FFFFH
Memory
64KB

MSX に使われている Z80 CPU には、基本的に 64 キロバイトのメモリーを接続できる。16 ビットのアドレスバスの信号で、0000H ～ FFFFH までのメモリーの中の 1 バイトを指定できるわけだ。またデータをやり取りするためのデータバスは、8 ビットになっている。

turbo R 以前の MSX の CPU である "Z80" は、8 ビット (物理的には 8 本の電線) のデータバスと、16 ビットのアドレスバスを持っている。これにより、64 キロバイトのメモリーを 1 バイトずつ読み書きできるわけだ。このような CPU を "8 ビット CPU"、8 ビット CPU が組み込まれたコンピューターを "8 ビットコンピューター" と呼ぶ。だから MSX は 8 ビットコンピューターというわけ。

さて、アドレスバスに関して具体的に説明すると、16 ビットのアドレスバスで指定できるメモリーの番地は、2 進数の

```
0000000000000000B
```

から (B は 2 進数を表わす記号)

```
1111111111111111B
```

まで。これを 10 進数で表わすと 0～65535 まで、16 進数で表わすと 0000H～FFFFH 番地までということになる (H は 16 進数を表わす記号)。各番地の内容は 8 ビット (=1 バイト) で、10 進数では 0～255 までの値を表わす。また、バイト単位で表わされたこれらのメモリーを、キロバイト単位に直すと 64。ゆえに 8 ビットコンピューターには、64 キロバイトのメモリーを接続できるというわけだ。

MSX は 8 ビットコンピューターだと前に書いたけど、最近は 16 ビットコンピューター (16 ビット CPU を搭載したコンピューター) も普及してきている。この場合はデータバスが 16 ビットなので、8 ビットコンピューターに比べ 2 倍の情報を 1 度に読み書きできる。アドレスバスも多くなり、それだけ多くのメモリーを接続できるなどの利点も多い。けれど配線が複雑になることなどから、比較的高価になってしまうのが現状だ。また、大型コンピューターの多くは、32 ビットや 64 ビットのデータバスとアドレスバスを持っている。

なお、MSX turbo R には 16 ビット CPU の R800 が搭載されたが、従来のカートリッジを接続できるように、データバスは 8 ビットのままだ。

2.1.3 メモリーの種類は働きによってイロイロ

メモリーには多くの種類がある。まず、部品の種類によって分類されるのが**"ROM"** と **"RAM"** 。ROM(Read Only Memory)とは内容を書き替えられないかわりに、電源を切っても内容が残るメモリーのこと。MSXの本体に内蔵されたソフトウェア(BASICなど)や、カートリッジで供給されるソフトウェアは、すべてこのROMの中に書き込まれているわけだ。またMSX2+になって標準装備された漢字ROMとは、漢字の文字の形を書き込んだ専用のROMのこと。

RAM(Randum Access Memory)とは、内容を自由に書き替えられるけど、電源を切るとその内容が消えてしまうメモリー。プログラム中で計算結果を一時的に記憶させたり、フロッピーディスクからプログラムを読み込んで実行させたりするのに利用する。たとえば、Mマガに載ったショートプログラムを打ち込んでゲームをするなんて場合も、このRAMに記憶されるわけだ。

なお、**"SRAM"** とは、消費電力が小さいRAMで、電池で動くノートパソコンやポータブルワープロ、そして、MSXやファミコンのバッテリーバックアップ付きゲームカートリッジにも使われているものだ。

このほかにも、メモリーを使い方によって分類することも可能。図2.1のように、CPUに直結しているメモリーは **"主記憶"** または **"メインメモリー"** 。つまり、MSX本体の漢字ROM以外のROMと、64キロバイトのメインRAMは、MSXのメインメモリーというわけだ。

またMSXには、このほかに **"ビデオRAM(VRAM)"** というメモリーもある。ビデオRAMとは、テレビ画面に表示する図形や文字を記憶するためのRAM。コンピューターの機種によっては、ビデオRAMがCPUに直結しているものもあるけど、MSXではVDP(ビデオ・ディスプレー・プロセッサーの略)という専用の部品を経由して、CPUとビデオRAMが接続されている。

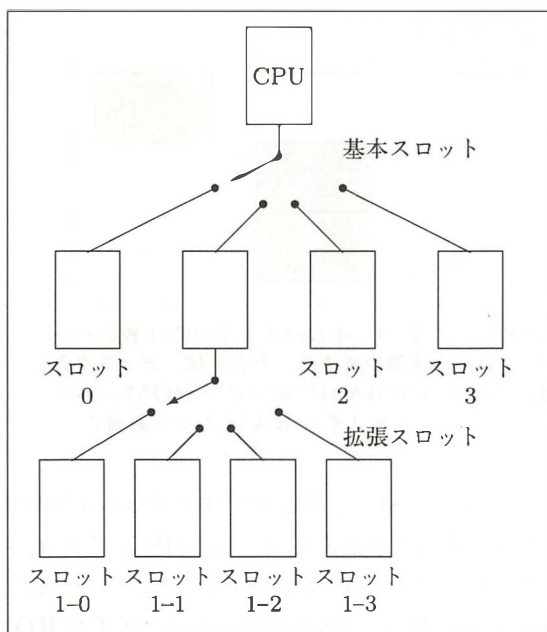
ここまでの基本的な話は、MSXに限らず、コンピューターについてのもっとも基本的な知識。MSXに付属してくるBASICの入門書などにも、詳しく書かれていると思うから参考にしよう。

2.1.4 MSXのスロットってどんなものなの？

はじめにも書いたように、8ビットCPUに接続できるメインメモリーは64キロバイト。しかし、これで話が済んでいたのは、初期の8ビットコンピューターだけ。最近はいろいろな方法を使って、64キロバイトを越えるメモリーを接続できるようになっている。

MSXの場合は"スロット切り替え"という方法。図2.2のように、64キロバイトのメモリーを4組用意し、それぞれを切り替えて使うことで、最大256キロバイト

のメモリーを扱うことが可能になる。これらのメモリーは **"基本スロット"** と呼ばれ、マシンに用意されたカートリッジスロットなどにも割り当てられている。

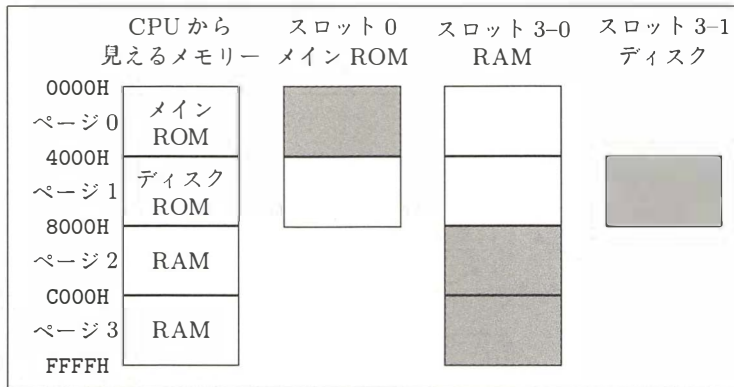
図 2.2: MSX のスロット構成 (その 1)

MSX では、64 キロバイトを越えるメモリーを扱うために、"スロット切り替え" という方法を使う。4 組の 64 キロバイトのメモリーを切り替えて、最大で 256 キロバイトのメモリーを扱うわけだ。この 4 組のメモリーを "基本スロット"、そこから拡張されるそれぞれ 4 個のスロットを、"拡張スロット" と呼ぶ。

さらに、1 個の基本スロットの代わりに 4 組の **"拡張スロット"** を切り替えて使う方法もある。この場合には、64 キロバイトのメモリーが全部で 16 組。つまり最大で 1 メガ (1024 キロ) バイトのメモリーを接続できるというわけ。ただし、拡張スロットをさらに拡張することはできない。

さて、スロットを切り替えることで、64 キロバイトを越えるメモリーを扱えるのはいいのだけど、その際にメモリー全体が同時に切り替わってしまうのは不便だ。そこで MSX では、メモリーを **"ページ"** という単位に分割して、扱うように考えられている。

メモリーの 0000H~3FFFH 番地までの 16 キロバイトをページ 0 、4000H~8000H 番地までの 16 キロバイトをページ 1 。同様にして、8000H~BFFFH 番地までをページ 2 、C000H~FFFFH 番地までをページ 3 という具合。それぞれ 16 キロバイトを 1 ブロックとしたページごとに、べつべつのスロットを選択できるわけだ。

たとえば、BASIC のディスク入出力関係の命令が処理されているときには、ページ 0 が BASIC インタープリターのメイン ROM 、ページ 1 がディスクインターフェースの ROM 、ページ 2 と 3 がメイン RAM に切り替えられる (図 2.3 参照)。

図 2.3: MSX のスロット構成 (その 2)

64 キロバイトのメモリーのアドレスは、16 キロバイトずつの 4 枚のページに分割され、各ページごとにスロットを選択できる。たとえば、ディスク入出力の命令を処理する場合は、ページ 0 が BASIC のメイン ROM 、ページ 1 がディスクの ROM 、ページ 2 と 3 がメイン RAM という具合だ。

BASIC インタープリターというのは、BASIC で書かれたプログラムを処理するプログラムのことで、MSX 本体の ROM に書き込まれている。MSX1 では 32 キロバイトの ROM に入っていたけれど、MSX2 では 48 キロバイト。そこで MSX2 では ROM が 2 個に分けられ、MSX1 と共通の部分が 32 キロバイトの **"メイン ROM"** に、MSX2 で拡張された機能が 16 キロバイトの **"サブ ROM"** に書き込まれている。

また、ディスク内蔵型の MSX や、外部ドライブのインターフェースカートリッジには、16 キロバイトの ROM が内蔵。BASIC のディスク入出力を処理するためのプログラム (DISK–BASIC) が、書き込まれているというわけ。だから、ディスク入出力中には、図 2.3 のような状態になるわけだ。

2.1.5 MSX の拡張性の秘密はスロットにあった

MSX のスロットは、メモリーを増設するだけでなく、MSX の機能を拡張するためにも使われる。ゲームカートリッジを接続するのも、モデムカートリッジを接続するのもスロットというわけ。

いま書いたように、MSX にディスクインターフェースカートリッジを接続すると、カートリッジ内の 16 キロバイトの ROM がスロットに接続される。そしてディスク入出力が行なわれるときには、自動的にメモリーがディスクインターフェース ROM のスロットに切り替えられるわけだ。このため、ディスクインターフェース自体が本体に内蔵されていても、カートリッジとして接続されていても、プログラ

ムの動作には支障がない。

また、ディスクインターフェースを接続すると、“CALL FORMAT” という命令が、通信カートリッジを接続すると、“CALL TELCOM” という拡張 BASIC の命令が、使えるようになる。これらの拡張命令は、カートリッジ内の ROM によって処理されるわけだ。MSX 以外の多くのパソコンでは、周辺機器を使うときにそれらを制御するプログラムをディスクから読み込む必要がある。ところが MSX では、インターフェースカートリッジを接続するだけで、自動的に BASIC の命令が拡張されるのだ。

また、MSX–DOS を機能アップした “日本語 MSX–DOS2” や、統合化ソフトとして話題の “HALNOTE” 、そして turbo R 専用の GUI(グラフィカル・ユーザー・インターフェース) として登場した “MSXView” も、カートリッジで供給されるソフトのひとつ。このように、ROM カートリッジをスロットに接続すると簡単に機能を拡張できることが、ほかのパソコンにない MSX の長所なのだ。

スロットは便利な機能だけど、それを使いこなしたプログラムを開発するのは、なかなか大変なこと。Z80 CPU のマシン語プログラムを自在に書けるプログラマーにとっても、スロットの概念を本当に理解するには 1 年以上かかるかもしれない。

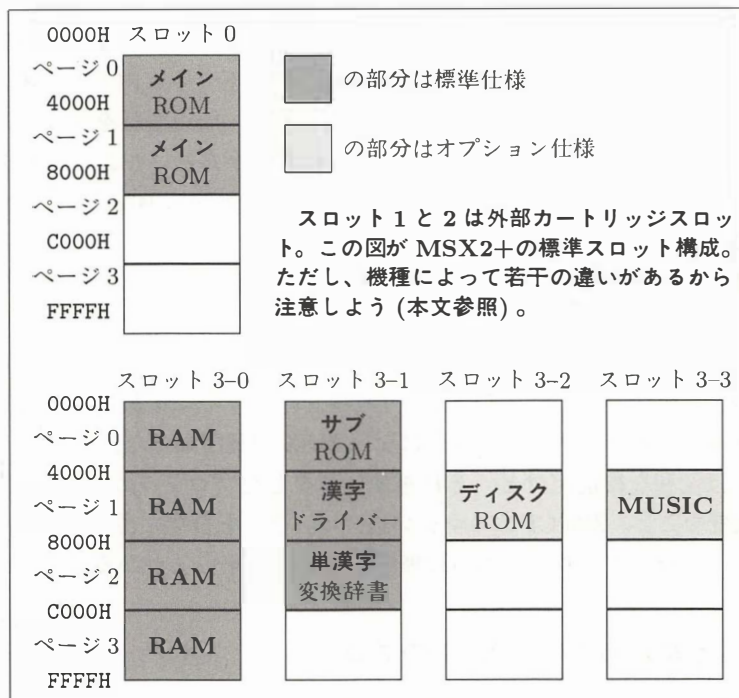
2.1.6 こう変わった MSX2+のスロット

いままで書いてきたように、MSX マシンに豊富な拡張性を持たせ、特徴づけてくれたのがスロットというもの。けれども、このスロットはまた、MSX の弱点でもあった。それが、“従来の MSX では機種によってスロット構成が異なるために、ソフトウェアの互換性の問題が起こりやすい” ということだ。

たとえば、スロット 1 と 3 がカートリッジスロットに割り当てられている機種で、ある特定のソフトが動かない。また、スロット 3 の拡張スロットに RAM が置かれていると、DOS からサブ ROM の機能を使えなかったりするなど。慎重にプログラムを作って、すべての MSX マシンについて動作を確認すれば、こうした問題は避けられるはず。けれども、あらゆるスロット構成のマシンに対応させると、プログラムが長くなったり、実行速度が遅くなったりという弊害も出てくる。一筋縄ではいかないのがスロットというわけだ。

これが MSX2+になって、やっとスロット構成に関するある程度の基準が決められた。図 2.4 と図 2.5 に示したのが、その MSX2+のスロット構成の例。本体に内蔵するソフトウェアの数によって、スロット 3 のみを拡張する場合と、スロット 0 とスロット 3 を拡張する場合とに分けられている。

図 2.4 に掲載したのが、スロット 3 のみを拡張する場合。基本スロット 0 に、BASIC のメイン ROM を置き、スロット 1 とスロット 2 を外部カートリッジスロットとする。

図 2.4: MSX2+のスロット構成の例 (スロット 3 のみを拡張する場合)

そして、スロット 3 の拡張スロットの**どれかひとつに**、64 キロバイトの RAM(メイン RAM) を、0~3 ページまでかならず同じスロットに RAM がくるように置く。同様に、サブ ROM 、漢字ドライバー、単漢字変換辞書の合計 48 キロバイトの ROM を、スロット 3 の拡張スロットの**どれかひとつに**置くという具合。図ではスロット 3 の 0(基本スロット 3 の 0 番目の拡張スロット) に RAM が、3 の 1 に ROM が置かれているけど、これは機種によって異なるわけだ。

これに対し図 2.5 は、スロット 0 とスロット 3 の両方を拡張する場合。基本スロット 0 の拡張スロット 0 に、メイン ROM が置かれている。スロット 3 の構成は、図 2.4 の場合とほぼ同じ。ディスクインターフェースを内蔵する場合には、スロット 0 の拡張ではなく、かならずスロット 3 の拡張に ROM が置かれることに注意しよう。

なお、図 2.4 と図 2.5 の薄い灰色の部分。つまり、ディスクインターフェース、MSX–MUSIC(FM 音源) 、通信、連文節変換辞書の各 ROM に関しては、MSX2+ のオプション仕様となっている。そのため、本体に内蔵されずに、外部カートリッジとして接続されることもある。

さて、図 2.5 と同じだけの内蔵ソフトウェアを持つ MSX2+には、理論上 36 通り

図 2.5: MSX2+のスロット構成の例 (スロット 0 と 3 を拡張する場合)

	スロット 0–0	スロット 0–1	スロット 0–2	スロット 0–3
0000H ページ 0	メイン ROM			
4000H ページ 1	メイン ROM	MUSIC	通信	連文節 変換辞書
8000H ページ 2				
C000H ページ 3 FFFFH				

	スロット 3–0	スロット 3–1	スロット 3–2	スロット 3–3
0000H ページ 0	RAM	サブ ROM		オプション
4000H ページ 1	RAM	漢字 ドライバー	ディスク ROM	オプション
8000H ページ 2	RAM	単漢字 変換辞書		オプション
C000H ページ 3 FFFFH	RAM			オプション

(濃い灰色) の部分は標準仕様　　(薄い灰色) の部分はオプション仕様

MSX2+本体に通信ソフトなどが内蔵される場合には、このようにスロット 0 と 3 が拡張される。スロット 3–3 には、メーカー独自のアプリケーション (ワードプロセッサーなど) が組み込まれることがある。

のスロット構成が考えられる。「うわー、そんなにあるのか」なんて声も聞こえてきそうだけど、これでも MSX2 のスロット構成よりは、組み合わせの数が減っているのだ。また、サブ ROM と漢字ドライバーと単漢字変換辞書がかならず同じスロットに置かれることで、MSX2+の漢字入出力が予想よりも速くなったはずだ。

2.1.7 スロットを拡張しちゃえ

MSX マシンにひとつかふたつ用意された外部カートリッジスロット (普段ゲームカートリッジを差し込むところ) は、どれも基本スロット。そこで、**“スロット拡張器”** を接続して、4 個の拡張スロットに拡張することができる。たとえば本体の 2 個

のスロットの両方に、スロット拡張器を接続すれば、合計8個ものスロットが誕生するわけだ。

ただ注意しなくてはいけないのが、拡張スロットでは動かないカートリッジもあるということ。日本語MSX–DOS2(RAMを内蔵したタイプ)なども、そのひとつ。使いたいソフトを確認してから拡張しよう。

スロット拡張器の製品名は“MSX拡張スロットボックス・EX–4”。価格2万9800円[税別]で、日本エレクトロニクス(電話03–3486–4181)から現在も発売されている。このほかにも、いくつかのMSXメーカーがスロット拡張器を発売していたが、現在ではほとんど入手不可能の状態だ。

2.2 スロット切り替えに挑戦

MSX を語るのに、避けてはとおれないのがスロットの概念。ここではソフトウェアでスロットを制御する方法や、MSX2+での仕様の変更点などを中心にお送りする。

2.2.1 スロットを切り替えるには

MSX におけるスロットの意味は説明したけれど、スロットを切り替える方法は説明していなかった。そこでここからは、その方法を紹介しよう。もちろん "人間がスイッチで切り替える" なんてヤボなものではなく、プログラムで切り替えられるのだ。

まず、MSX の CPU である Z80 には、**"I/O ポート"** というものがある。これは、CPU が外部 (つまり VDP や FM 音源などの周辺機器) と通信するための、電話線のようなものだ。Z80 には合計 256 本もの I/O ポートがあるけれど、それぞれに 0~255(16 進数では 00H~FFH) までの番地を割り当てることで、区別している。

BASIC でこれらのポートを扱うには、"INP" 関数で I/O ポートから 1 バイトの値を読み、"OUT" 命令で 1 バイトの値を書く。またマシン語では、"IN" 命令と "OUT" 命令がこれと同じ働きをする。

さて、この I/O ポートの A8H 番地に書き込まれた値によって、基本スロットが切り替わる。逆にこの番地の値を読むと、現在のスロットの状況がわかるわけだ。ビット 7 と 6 がページ 3 、5 と 4 がページ 2 、3 と 2 がページ 1 、1 と 0 がページ 0 というように対応しているので、11110000B(B は 2 進数を意味する) という値を書き込むと、ページ 3 と 2 がスロット 3 に、ページ 1 と 0 がスロット 0 に切り替わるぞ。また、拡張スロットを切り替えるにはメモリーの FFFFH 番地を使うのだけど、こちらは複雑なのでここでは省略する。

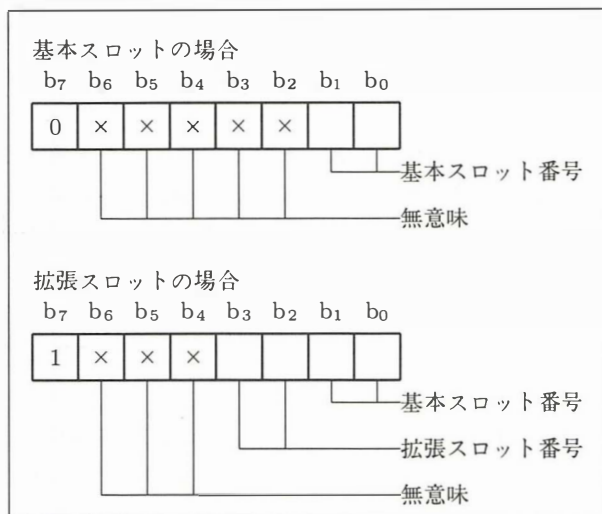
ところで、プログラムによって直接スロットを切り替えると、面倒なだけではなく、機種によってはプログラムが動かないといった、互換性の問題が起こりやすい。そこで実際には、**"BIOS"** によってスロットを切り替える。BIOS とは、**"Basic Input Output System"** という意味。あとでくわしく説明するけど、ハードウェアを制御するための、マシン語サブルーチンの集まりだ。これにはスロットを切り替える以外にも、多くの機能があるのでチェックしよう。

2.2.2 スロット番号の指定方法

BIOS を使ってスロットを切り替える場合には、基本スロット番号と拡張スロット番号を、それぞれべつに指定すればいい。でもそのためには、2 個のレジスター (CPU 内のデータの一時記憶場所) が必要になってしまい不経済 (?) だ。そこで、図

2.6のように8ビット(1バイト)の各ビットをうまく使って、基本スロットと拡張スロットをまとめて指定する方法が取られている。

たとえば基本スロット0を指定するには、00000000B(16進数では00H)という値を、基本スロット3の拡張スロット1を指定するには、10000111B(87H)という値を指定すればいい。

図 2.6: スロット番号の指定方法

スロット番号は、図のように8ビット(=1バイト)の値で表わされる。たとえば、基本スロット番号の0を指定するためには00000000B、基本スロット3の拡張スロット1を指定するためには10000111Bという値を使えばいいわけだ。なお、図の中で、×がつけられた各ビットの内容は無視される。

2.2.3 スロットを操作するBIOSの機能

まず、BIOSの機能を書き表わすための記号を覚えよう。[E]とはBIOSを呼び出す前に値を設定すべきレジスター。[R]はBIOSが値を返すレジスター。[M]はBIOSが無意味な値を書き込む、つまり元の内容が壊されるレジスターを表わす。またIYHとは、IYレジスターの上位バイトを表わし、下位バイトの内容は無視される。またはじめに書かれた番地は、そのBIOSを呼び出すためのエントリーアドレスだ。

RDSLT 000CH 番地

[機能] Aレジスターで指定されたスロットの、HLレジスターで指定された番地の内容を読む。

[E] A スロット番号
HL 番地

R　A 読んだ値

M　AF 、BC 、DE

注　割り込みが禁止される。

WRSLT　0014H 番地

機能　A レジスターで指定されたスロットの、HL レジスターで指定された番地に、E レジスターの内容を書き込む。

E　A スロット番号
HL 番地
E 書き込む内容

R　なし

M　AF 、BC 、D

注　割り込みが禁止される。

CALSLT　001CH 番地

機能　ほかのスロットにあるサブルーチンを呼び出す。

E　IX 呼び出す番地
IYH スロット番号

R　呼び出す相手による

M　IX 、IY 、裏レジスター

注　現在のスロットの状態をスタックに保存し、目的のサブルーチンをコールする。AF 、BC 、DE 、HL レジスターの内容は、そのままサブルーチンに渡され、サブルーチンが RET 命令を実行すると、元のプログラムに戻る。このときも、AF 、BC 、DE 、HL レジスターの値はサブルーチンから渡される。何バイトのスタックが使われるかどうかは、スロット構成によって異なる。

ENASLT　0024H 番地

機能　スロットを切り替える。

E　A スロット番号
H ページ (上位 2 ビット)

R　なし

M　AF 、BC 、DE 、HL

注　たとえば、2ページを切り替えるためには、Hレジスターに80H～BFHの値を設定すればよい。割り込みが禁止される。

CALLF　0030H番地

機能　ほかのスロットにあるサブルーチンを呼び出す。

E　以下のプログラムのように、"RST 30H"命令に続けてスロット番号と番地をプログラムに書き込んでおく。

```
RST   30H
DB    スロット番号
DW    番地
```

R　呼び出す相手による

M　IX、IY、裏レジスター

注　スロットと番地の指定方法以外は、CALSLTと同じ。特別な目的(フック)に使う。

EXTROM　015CH番地

機能　サブROMを呼び出す。

E　IX 呼び出す番地

R　呼び出す相手による

M　IX、IY、裏レジスター

注　自動的にサブROMのスロットが選択される以外は、CALSLTと同じ働きをする。

と、以上紹介してきたBIOSには若干の制限がある。どれもページ3に対しては使えない。ページ0に対してはDOSからメインROMを呼び出す場合にのみ使える。ページ2と3に対しては問題なく使える、ということだ。"使えない"といっても、スロット構成によっては使えることもあるから、"自分のMSXだけで動くプログラム"を作らないように注意しよう。とくにDOSからサブROMを呼び出そうとしてCALSLTを使うと、スロット構成とディスクインターフェースの種類によって、動いたり動かなかったりするぞ。

2.2.4　スロット構成を知る方法

前にも説明したように、MSXのスロット構成は機種によって異なる。ディスクインターフェースのようなオプション仕様もあるため、マシンの数だけスロット構成

があるといっても過言ではない。そこで、自分のマシンのスロット構成と、オプション機器の有無を調べる方法を紹介しよう。

メモリーの F380H~FFFEH 番地までを **"システムワークエリア"** といい、ここには BIOS などにとって重要な情報が記憶されている。ディスクインターフェースが接続されれば、システムワークエリアより少し番地が小さい場所に **"ディスクワークエリア"** が用意されるわけだ。またスロットに関係する情報は、表 2.1 のようにシステムワークエリアとディスクワークエリアに記憶されている。

メイン RAM がどのスロットにあるかという問題は重要だけど、表 2.1 の "RAMAD0" などはディスクワークエリア内にある。そのため、ディスクがないとこれらの情報はわからないという問題もある。

リスト 2.1は、これらのシステムワークエリアから調べたスロット構成を、わかりやすく表示するプログラムだ。機種によって構成が違ってくるから、自分の MSX でも試してみよう。

表 2.1 に掲載した以外にも、プログラムの役に立つシステムワークエリアがあるけど、詳細は "MSX2 テクニカルハンドブック" などを見てほしい。それから、これらのシステムワークエリアは、とくに指示される場合を除いて、アプリケーションプログラムが書き替えてはいけない。メモリーが不足して苦しまぎれにシステムワークエリアを使うプログラムがあるけど、互換性をなくするもとなので注意しよう。

表 2.1: スロットに関するシステムワークエリア

名称	番地	意味
RAMAD0	F341H	ページ 0 の RAM のスロット番号 (1)
RAMAD1	F342H	ページ 1 の RAM のスロット番号 (1)
RAMAD2	F343H	ページ 2 の RAM のスロット番号 (1)
RAMAD3	F344H	ページ 3 の RAM のスロット番号 (1)
MASTER	F348H	ドライブ A のインターフェースのスロット番号 (1)
EXBRSA	FAF8H	サブ ROM のスロット番号 (MSX1 では 0)
EXPTBL	FCC1H	メイン ROM のスロット番号
	FCC2H	スロット 1 が拡張されているかどうか (2)
	FCC3H	スロット 2 が拡張されているかどうか (2)
	FCC4H	スロット 3 が拡張されているかどうか (2)

(1) ディスクがある場合のみ有効。
(2) 拡張されていれば 80H 、そうでなければ 0 。

2.2.5 システムワークエリアを探ってみる

MSX2 用のゲームソフトの中に、MSX2+で動かせば SCREEN 12 のタイトル画面を表示するようなものがある。また、モデムカートリッジがないのに通信しよう

とすると、親切にエラーメッセージを表示するプログラムもある。こんなソフトを作るために、プログラムがハードの種類や構成を調べる方法を紹介する。

まず **“ディスクがあるかどうか”** を調べるためには、FFA7H 番地の内容を読む。もし C9H であればディスクがなく、そのほかの値であればディスクがある。

“MSX の種類” を調べるためには、メイン ROM の 2DH 番地を読む。0 ならば MSX1、1 ならば MSX2、2 ならば MSX2+、そして 3 ならば turbo R というわけ。

一般に、2BH 番地と 2CH 番地の内容は 0 だけど、**“海外への輸出用に作られた MSX”** では、キーボードや通貨記号の種類を表わす番号が入っている。輸出用ソフトウェアを作る場合だけ気にすればいいので、番号の一覧は省略する。

これは余談になるけど、MSX パソコンはヨーロッパをはじめ、ソビエトや、中近東のクウェートなどにも相当数が輸出されている。また、お隣の韓国では、学校に多数導入され、授業に役立てられているとか。なんともインターナショナルなマシンなのだ。

さて、**“ビデオ RAM 容量”** を調べるには、FAFCH 番地を読む。ビット 2 とビット 1 の値が、00 ならば 16 キロバイト、01 ならば 64 キロバイト、10 ならば 128 キロバイトだ。これ以外のビットはべつの目的に使われているようなので無視しよう。次のページのリストのように “AND 6” でビット 2 とビット 1 の値を取り出し、それを 2 で割ればいい。

このリストでは、おまけとして **“拡張 BIOS”** の有無も調べている。これは、通信モデム、FM 音源、漢字辞書といった、オプションハードウェアを制御するための機能だ。FB20H 番地のビット 0 の内容が 1 で、FFCAH 番地の内容が C9H でなければ、何らかの拡張 BIOS 機能があることになる。それが何であるか調べるには、複雑なマシン語のプログラムが必要になるので、今回はパス。それから、これは仕様書には書かれていないのだけど、FFCBH 番地の内容は、拡張 BIOS 機能を持っているプログラムのスロット番号らしい。

なお、意味が決められていないビット、たとえばスロット番号を表わす値の、ビット 6 からビット 4 などには、何が書き込まれているかわからない。そこで、“AND” を使って、その内容を無視していることがわかるかな。

何度も書くようだけど、たとえ同じメーカーのマシンであっても、機種によってスロット構成が異なることがある。だから、自分の持っているマシンで試したあとは、友だちのマシンでも試してみよう。多くのマシンでテストして、その結果を表にしてみるとおもしろいぞ。

リスト 2.1 (WHO_AM_I.BAS)

```
100 ' Analizing slot structure of MSX
110 ' by nao-i on 9. Jan. 1989
120 CLEAR : DEFINT A-Z : CLS
130 VE = PEEK(&H2D) : ' Version No. of BASIC
140 IF VE=0 THEN PRINT "I am MSX1"
150 IF VE=1 THEN PRINT "I am MSX2"
160 IF VE=2 THEN PRINT "I am MSX2+"
165 IF VE=3 THEN PRINT "I am MSX turbo R"
170 IF VE>3 THEN PRINT "Who am I ?"
180 VR = (PEEK(&HFAFC) AND 6) ¥ 2 : ' size of VRAM
190 IF VR=0 THEN PRINT "VRAM 16KB"
200 IF VR=1 THEN PRINT "VRAM 64KB"
210 IF VR>1 THEN PRINT "VRAM 128KB"
220 MR = PEEK(&HFC49) : ' size of main RAM
230 IF MR >= &HE0 THEN PRINT "RAM 8KB"
240 IF MR < &HE0 AND MR >= &HC0 THEN PRINT "RAM 16KB"
250 IF MR < &HC0 THEN PRINT "RAM >= 32KB"
260 FOR SS = 0 TO 3 : 'EXPTBL
270   PRINT USING "Slot # is "; SS;
280   FF = PEEK(&HFCC1 + SS) AND 128
290   IF FF THEN PRINT "expanded slot" ELSE PRINT "primary slot"
300 NEXT SS
310 PRINT
320 SS = PEEK(&HFCC1) : PRINT "Main ROM is in "; : GOSUB 530
330 SS = PEEK(&HFAF8) : PRINT "Sub  ROM is in "; : GOSUB 530
340 IF PEEK(&HFFA7) <> &HC9 THEN 360
350 PRINT "I have disk(s)." : GOTO 450
360 PRINT "I have no disk."
370 SS = PEEK(&HF348) : PRINT "FDC ROM is in "; : GOSUB 530
380 SS = PEEK(&HF341) : PRINT "P0  RAM is in "; : GOSUB 530
390 SS = PEEK(&HF342) : PRINT "P1  RAM is in "; : GOSUB 530
400 SS = PEEK(&HF343) : PRINT "P2  RAM is in "; : GOSUB 530
410 SS = PEEK(&HF344) : PRINT "P3  RAM is in "; : GOSUB 530
420 PRINT "Bottom address of disk work area is ";
430 PRINT RIGHT$("00"+HEX$(PEEK(&HFC4B)),2);
440 PRINT RIGHT$("00"+HEX$(PEEK(&HFC4A)),2)
450 ' detectiong extended BIOS
460 IF (PEEK(&HFB20) AND 1) = 0 THEN GOTO 520
470 IF PEEK(&HFFCB) = &HC9 THEN GOTO 520
480 PRINT : PRINT "I have extended BIOS."
490 SS = PEEK(&HFFCB)
500 PRINT "ROM of the extended BIOS may be in "
510 GOSUB 530
520 END
530 ' displaying slot number
540 PRINT USING "primary slot #"; SS AND 3;
550 IF (SS AND 128) = 0 THEN 570
560 PRINT USING " extended slot #"; (SS AND 12) ¥ 4;
570 PRINT : RETURN
```

2.2.6 MSX2+のハードウェア仕様

MSX2+には、ハードウェアの細かい改良点が加えられた。表 2.2 にまとめたのが、新しく仕様が定義または追加されたI/Oポートだ。

表 2.2: MSX2+の I/O ポート

I/O 番地	用途
7CH	本体内蔵 FM 音源
7DH	本体内蔵 FM 音源
DAH	第 2 水準漢字 ROM
DBH	第 2 水準漢字 ROM
F4H	初期化の制御
F5H	デバイスイネーブル

これが新しく仕様が定義または追加されたもの。ただし、実際のプログラムでは、I/O ポートを直接使わずに BIOS を使うほうがよい。

7CH 番地と 7DH 番地は、本体に内蔵された FM 音源を操作するための I/O ポート。これとは違い、カートリッジで供給されるタイプの FM 音源は (今後もし発売されるなら)、パナソニックの “FM–PAC” と同じ I/O ポートを使う。

FM 音源が本体に内蔵されているかどうかを調べるためには、各スロットについて、4018H 番地から 401FH 番地までを読む。その内容が “APRLOPLL” という文字列と一致すれば、そのスロットに FM 音源を制御するプログラムの ROM があり、本体に FM 音源が内蔵されているというわけだ。

一方、FM 音源カートリッジの場合には、4018H 番地からの内容が “PAC2OPLL” のように、製品の種類を表わす 4 文字と “OPLL” いう文字になるようだ。

“MSX–Write” や一部のモデムカートリッジには、最初のメニューで “BASIC” を選ぶと、リセットされたように MSX のタイトル画面が表われるものがある。これは、ソフトウェアの準備の都合で、メイン ROM の 0 番地へジャンプして、リセットと同じような処理をさせているからだ。

以前は、0 番地へのジャンプと本当のリセットを確実に区別する方法がなかったので、ソフトウェアの誤動作が起きることがあった。それが MSX2+からは、I/O ポートの F4H 番地にリセットの状態を調べるためのハードウェアが追加された。ただし、実際には、次のように MSX2+のメイン ROM に追加された BIOS を使う。

```
CALL 17AH
OR   80H
CALL 17DH
JP   0
```

初期化時に呼び出された ROM カートリッジのプログラムは、

```
CALL 17AH
```

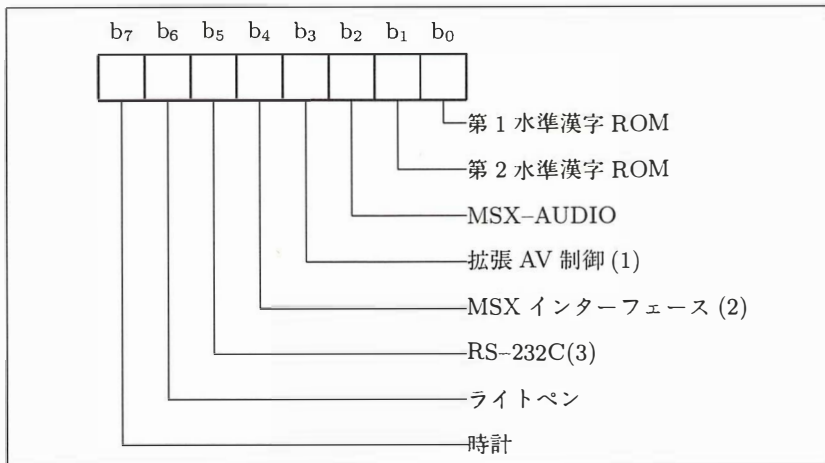
を行なう。そして、A レジスターのビット7が0ならば本当のリセット。1ならばジャンプ0で、自分が呼び出されたことがわかるというわけだ。

2.2.7 衝突を防ぐデバイスイネーブル

漢字 ROM を内蔵している MSX に漢字 ROM カートリッジを接続すると、漢字が正しく表示されないだけでなく、ハードウェアが衝突して故障する危険がある。これを防ぐありがたい機能が、I/O ポートの F5H 番地で制御される“デバイスイネーブル”というものだ。

図2.7 に書かれているハードウェアは、リセット時にバスから切り離されている。そして、I/O ポートの F5H 番地の1バイト (8ビット) の値を書き込むことで、1になったビットに対応する内蔵ハードウェアが、バスに接続されるわけだ。これらの処理は、リセットまたはジャンプ0 (ソフトウェアによって、メイン ROM の0番地にプログラムの実行が移ること) のあとで自動的に行なわれる。

MSX2 では、I/O ポートの F5H 番地に0を書き込んだ場合、既に接続されているハードウェアを切り離すかどうかの、規定がされていなかった。このため“MSX–

図 2.7: デバイスイネーブル

I/O ポートの F5H 番地により、本体内蔵のハードウェアを有効にするか無効にするかを選択する (1 を書き込まれたビットに対応するハードウェアが有効)。
(1) I/O ポートの F7H 番地で制御されるスーパーインポーズ機能など。
(2) 仕様書にあるが実用化されていない。
(3) モデムには関係ない。

Write" などが、ROM の 0 番地へジャンプして BIOS を再度初期化しようとすると、混乱が起こることもあったわけだ。

それが MSX2+からは、0 を書き込めば、内蔵ハードウェアがバスから切り離されるように統一された。これにより、I/O ポートの F4H 番地と F5H 番地を活用して MSX2+用の基本ソフトウェアを作ることで、互換性と信頼性が、いままで以上によくなるわけだ。

2.3 MSX trubo R のスロット構成

ここでは、新しく発表された、MSX turbo R のスロット構成について解説する。特筆すべきは、ここにきて、やっとのことで、スロット構成が統一されたことだ。これは、なんとも意義深いことなのだ。

2.3.1 ついにスロット構成が統一されたぞ

図 2.8 が、turbo R のスロット構成だ。CPU の高速化に対応し、アプリケーションプログラムの開発やデバッグを容易にするために、スロット構成が統一された。

この図では、スロット 3–0 に 64 キロバイトの RAM があるように見えるけど、実際にはメモリーマッパーをとおして、256 キロバイトのメイン RAM が接続されている。このうち 64 キロバイトを越える部分は、日本語 MSX–DOS2 のワークエリアや RAM ディスク、べつの章で説明する “DRAM モード” などに、通常は使われる。でも、アプリケーションプログラムが拡張 BIOS を使ってマッパーを切り替え、これらの RAM を使うことも可能だ。

また、スロット 3–2 のページ 1 には DOS のシステム ROM が収められている。といっても、ここには 16 キロバイトの DOS1(MSX–DOS) の ROM と、48 キロバイトの DOS2 の ROM が接続されていて、必要に応じて自動的に切り替えられるようになっている。

標準化されたこのスロット構成の最大の利点は、DOS のプログラムが普通の方法でサブ ROM をインタースロットコールできることと、DOS の割り込み処理プログラムをどの番地に置いてもよいことだ。昔の M マガで紹介したことがあるのだけれど、拡張されたスロット 0 に RAM とサブ ROM があった場合は、MSX–DOS のインタースロットコール機能と、割り込み処理プログラムが暴走する可能性があった。しかし turbo R では、拡張されたスロット 3 に RAM とサブ ROM があるので、このような問題が起こらないわけだ。

また、OPLL ドライバー、つまり FM–BIOS の ROM は、かならずスロット 0–2 に配置されている。そのため turbo R 専用ソフトウェアは、FM–BIOS があるスロットを探す手順を省略してもいい。

このほか、スロット構成が統一されたことによる利点の特殊な例として、“コナミの 10 倍カートリッジ” があげられる。これはスロット 1 に 10 倍カートリッジを、スロット 2 にゲームカートリッジをセットして使う必要があったもので、一部の MSX1 と MSX2 マシンでは動作しなかった。ところが、MSX2+と turbo R では、外部スロットがスロット 1 と 2 に決められたので、こうした特殊なプログラムも、簡単にかつ確実に実現できるような環境が整ったわけだ。

図 2.8: MSX turbo R のスロット構成

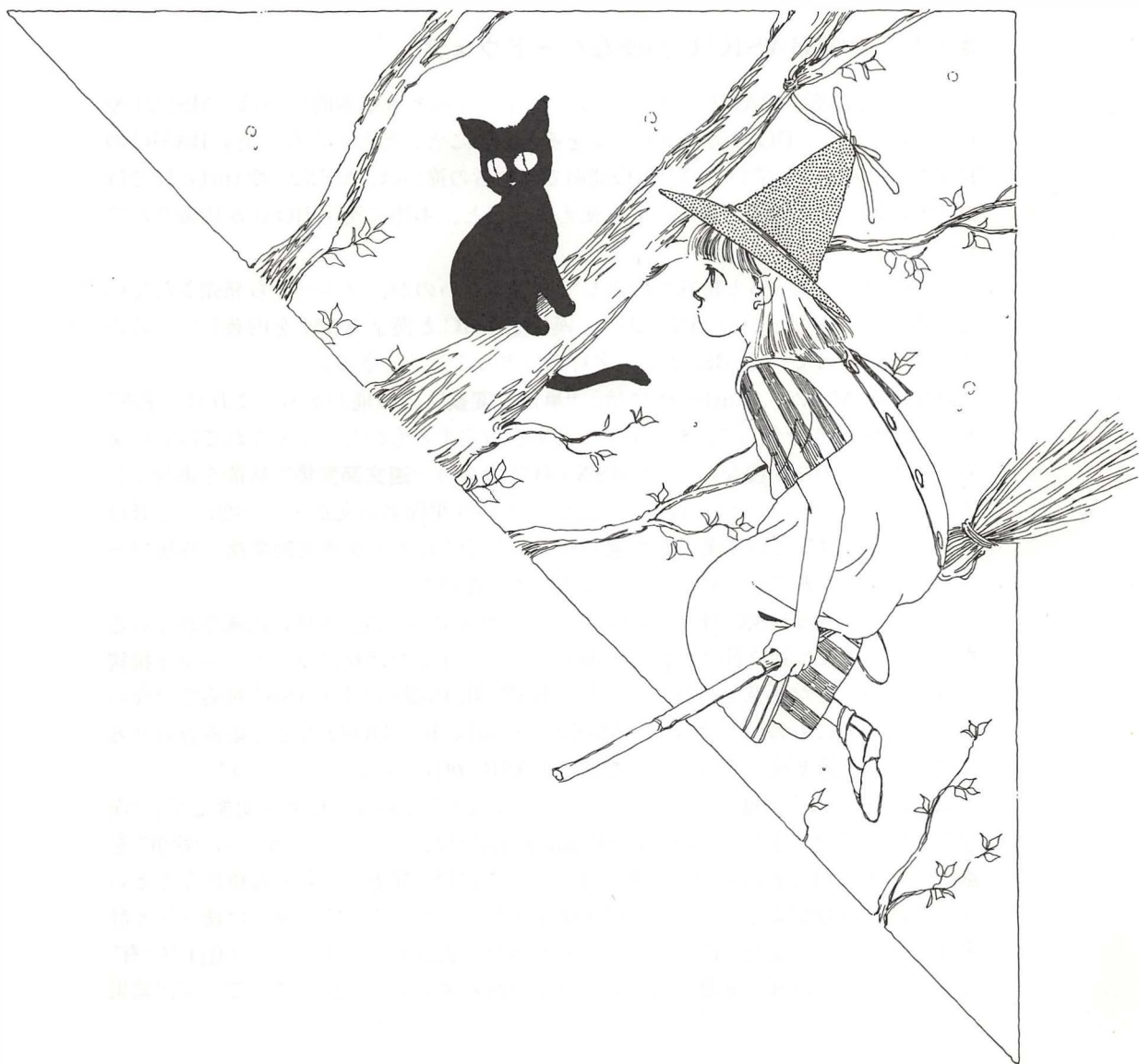
スロット 0–0
スロット 0–1
スロット 0–2
スロット 0–3
0000H
ページ 0
4000H
ページ 1
8000H
ページ 2
C000H
ページ 3
FFFFH
メイン ROM
メイン ROM
MUSIC

スロット 1
スロット 2
0000H
ページ 0
4000H
ページ 1
8000H
ページ 2
C000H
ページ 3
FFFFH
外部スロット
外部スロット

これまでは、いくつかのパターンが存在したスロット構成も、turbo R ではこのように標準化された。(注 1) のメイン RAM という部分には、メモリーマッパーを使って 256 キロバイトの RAM が接続される。(注 2) の部分は MSX–DOS1(16 キロバイト) と、MSX–DOS2(48 キロバイト) のふたつが、状態に応じて自動的に切り替えられるようになっている。

スロット 3–0
スロット 3–1
スロット 3–2
スロット 3–3
0000H
ページ 0
4000H
ページ 1
8000H
ページ 2
C000H
ページ 3
FFFFH
RAM (注 1)
サブ ROM
漢字ドライバー
DOS (注 2)
メーカーオプション

またソフトウェアハウスにとっては、スロット構成が統一されたために、特定のスロット構成で発生するバグに悩まされることが減るのが、最大の利点といえる。ソフトウェアを作る立場からすると、CPU の高速化や RAM 容量が増設されたことよりも、スロット構成の統一のほうがはるかにウレシイのだ。turbo R 万歳！

第3章 漢字BASIC

この章は、MSXマガジン1989年4月号の“MSX2+テクニカル探検隊”の記事を再編集したものである。

3.1 漢字BASICを解析

MSX2+以降のマシンやDOS2の特長のひとつは、漢字が使いやすくなったこと。BASICのプログラムの文字列やファイル名にも、漢字を使うことができる。この章では、その漢字BASICの機能をレポートする。

3.1.1 漢字BASICに必要なハードウェア

どうしたら漢字BASICが使えるか。まず、もっとも基本的なのは、MSX2+やturbo R本体か、DOS2カートリッジをそろえること。どちらにも、漢字BASICのROMが組み込まれているのだ。機能面での両者の違いは、MSX2+やturbo RではSCREEN 10~12の自然画モードを使えることと、本体に漢字ROMが内蔵されていること。

このほかに、ちょっと特殊な例として注目したいのが、ソニーから発売されている日本語カートリッジの“HBI–J1”。漢字BASICと漢字ROMを内蔵しているので、すでに持っているMSX2を漢字対応にすることができる。

DOS2とMSX2+、turbo Rには、**“単漢字変換”**の機能がある。これは“読み”や“JISコード”を使って、1文字ずつ漢字を指定するものだ。でもこれでは長い文章を入力するのに不便なので、**“MSX–JE”**という**“連文節変換”**機能を追加できる。たとえば“きょうはいいおてんきです”という平仮名の文章を、一度に“今日は良いお天気です”という漢字かな混じり文にしてくれるのが連文節変換。専用ワープロなどで使われているのと、同じ方式のものなのだ。

表3.1が、そのMSX–JEを内蔵したハードウェアの一覧。本体に内蔵されているなら、そのままで連文節変換機能を使えるし、そうでなければカートリッジを接続すればいい。ただ注意してほしいのは、MSX–JE内蔵=漢字BASIC対応ではないこと。はじめにも書いたように、MSX2+やturbo R、DOS2などと組み合わせることで、連文節変換をサポートした漢字BASICが使えるようになるのだ。

MSX–JEには学習機能がある。たとえば“かんじ”という平仮名を変換して、“漢字”と“幹事”と“感じ”の3種類の候補が表示されたとする。この中から“幹事”を選ぶと、次に同じ単語を変換しようとしたときには“幹事”が第1候補になるという仕組み。自分がよく使う単語の優先順位が上がっていくので、使えば使うほど辞書が自分に合って変換の能率がよくなるわけだ。表3.1の“SRAM”の項目が“有”となっているものは、電源を切っても内容が残るメモリーを使って、この学習結果

表 3.1: MSX–JE 内蔵ハードウェア一覧

メーカー	製品名	形態	漢字 ROM	SRAM
パナソニック	FS–A1ST	MSX turbo R	1、2	有
パナソニック	FS–A1WSX	MSX2+	1、2	有
パナソニック	FS–A1WX	MSX2+	1、2	有
パナソニック	FS–SR021	カートリッジ (1)	1、2	有
パナソニック	FS–4600F	MSX2	1	有
パナソニック	FS–PW1	プリンター (2)	1	有
ソニー	HB–F1XV	MSX2+	1、2	有
ソニー	HB–F1XDJ	MSX2+	1、2	有
ソニー	HBI–J1	カートリッジ	1、2	有
サンヨー	PHC–77	MSX2	1	無
HAL 研究所	HALNOTE	カートリッジ (3)	1、2	有
アスキー	MSX–Write	カートリッジ	1	無
アスキー	MSX–Write Ⅱ	カートリッジ	1、2	有

漢字 ROM の項目は、第 1 、第 2 水準漢字 ROM の有無。SRAM“有”は、電源を切っても学習が残るもの。(1) A1WX のワープロをカートリッジ化。(2) カートリッジと専用プリンターのセット。(3) カートリッジとディスクによる専用 OS 。

を残すようになっている。

3.1.2 MSX–JE 対応のソフトウェアとは

MSX–JE はただのワードプロセッサーや拡張 BASIC ではなく、いろいろなアプリケーションと組み合わせて、連文節変換の機能を使えるようになっている。漢字 ROM や連文節変換辞書の ROM などは比較的高価なので、MSX–JE 自体を多くのソフトウェアが共有することは、とっても経済的なのだ。

アプリケーションが MSX–JE を利用する方法は仕様書で定められているので、“MSX–JE 対応” と表示されているソフトウェアは、どの MSX–JE とでも組み合わせられるようになっている。ただし、表 3.1 の HALNOTE は、カートリッジとシステムディスクの組み合わせで “HALOS” という専用 OS を使うようになっているので、HALOS 用でないソフトウェアとの組み合わせには制限がある。

3.1.3 漢字ドライバーの動作原理を解説する

MSX に限らず、コンピューターの内部で漢字を表わす方法を、簡単に説明しておく。まず英字とカタカナは 1 バイト (8 ビット) の値で表わせるけれど、漢字を表わすためには 2 バイトの符号が必要になる。JIS(日本工業規格) では、次のように漢字

を2バイトの符号で表わしている。

亜……3021H(第1水準)

腕……4F53H(第1水準)

弌……5021H(第2水準)

龠……737EH(第2水準)

しかし、この **“JIS コード”** には、英字と漢字が混ざると処理が面倒になるという問題がある。そこで、JIS コードをコンピューターが処理しやすいように変換した **“シフト JIS コード”** が、MSX を含む多くのパーソナルコンピューターで使われている。一部では **“マイクロソフト漢字コード”** とも呼ばれているけど、これはまあ興味のある人だけ覚えておこう。

いずれにせよシフト JIS コードは、英文字用に作られたソフトウェアを、そのまま、あるいはわずかな修正で、日本語でもだまして使えるように設計された便利な漢字コード。16 ビットパソコンでもっとも普及しているオペレーティングシステムの MS-DOS や、OS-9 といった最近話題のオペレーティングシステムにも、シフト JIS 漢字コードが使われている。だから、異なるコンピューター間で文書ファイルを交換することも、できるというわけだ。

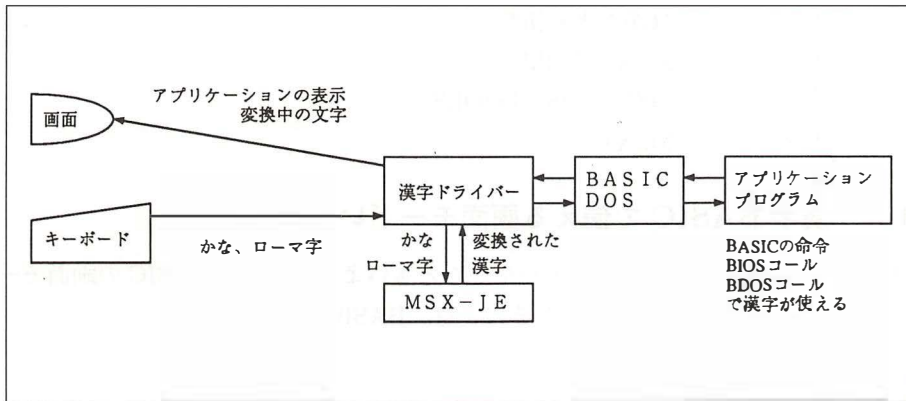
ところで、漢字コード表を見ていると、数字や英字が混じっていることに気づく。混乱を避けるために、2バイトの漢字コードで表わされる英字や漢字などを総称して **“全角文字”**。これに対し、1バイトの文字コードで表わされる文字を **“半角文字”** と呼び、両者を区別することにする。たとえば、半角文字の “ABCD” と全角文字の “ＡＢＣＤ” は、まったくべつの文字として扱われるから注意しよう。

それでは、いよいよ本題に入るぞ。MSX2+と turbo R、DOS2 の漢字入出力機能は **“漢字ドライバー”** と呼ばれている。これは、BASIC や DOS に限らず、多くのアプリケーションプログラムが利用できるようになっているものだ。

図 3.1 にあるのが、漢字ドライバーの動作原理。順を追って説明すると、まずキーボードから入力された全角の “かな” または “ローマ字” を読んで (判断して)、画面に表示する。次に MSX-JE を呼び出し、ひらがなを漢字 (全角文字) に変換。最後に、変換された漢字のコードが、BASIC または DOS をとおして、アプリケーションプログラムに送られるというわけ。

これとは逆に、アプリケーションプログラムが画面に文字を表示する場合は、漢字コードを漢字ドライバーに送ればいいわけだ。

この漢字ドライバーは、アプリケーションプログラムにとっては、BASIC の命令や BIOS コール、BDOS コール (DOS のプログラムが入出力を行なうための、BIOS に似た機能) などに、漢字入出力機能が追加されたように見えるだけ。とくに、アプ

図 3.1: 漢字ドライバーの動作原理

リケーションプログラム自身が MSX–JE の辞書を操作しなくてよいことが、漢字ドライバーの大きな利点だ。

3.1.4 JE 対応ハード&ソフト

ここでは参考までに、これまでに発売された MSX–JE 対応のハードとソフトをまとめてみた。

- モデムカートリッジ

パナソニック	FS–CM1
パナソニック	FS–CM820
ソニー	HBI–1200
キヤノン	VM–300
明星電気	V–3

- モデム内蔵 MSX

パナソニック	FS–A1FM
ソニー	HB–T7
ソニー	HB–T600
三菱	ML–TS2H

- ソフトウェア

ソニー	文書作左衛門
ソニー	はがき書右衛門
アスキー	MSX–TERM
アスキー	MSX–DOS2 TOOLS
アスキー	MSXView

3.1.5 漢字BASICで使える画面モードいろいろ

MSX2+やturbo Rの画面モードがやたらと多いように、漢字BASICの画面モードも複雑だ。表3.2に掲載したのがその一覧。BASICでは、

```
CALL KANJI
WIDTH
```

命令で。DOS2では

```
KMODE
MODE
```

命令で、それぞれ画面モードを指定するわけだ。たとえば、

```
CALL KANJI0 : WIDTH 32
```

という命令を実行すると、画面は32文字×12行表示になる。同じことをDOS2でやるには、

```
KMODE 0
MODE 32
```

とすればいい。ただし、英語版のシステムを立ち上げた場合は、漢字ドライバーが読み込まれていないから、一度BASICにもどって漢字モードを呼び出そう。

それでは、表3.2の意味を詳しく説明する。まず“画面ドット数”とは、画面に表示される点(ドット)の数。漢字はこの点の組み合わせで表わされる。

縦が424ドットの画面モードでは、“インターレース”という表示方法が使われている。これは縦方向に半ドットずらした2枚の画面を交互に表示し、結果として画面のドット数を増やす方法。ただ、画面がちらつくため、目が疲れやすいのが欠点だ。

“漢字ドット数”とは、1個の漢字を表わすための点の数。普通は漢字を16×16ドットで表わすけれど、横12×縦16ドットに圧縮して、横512ドットの画面に40文字の漢字を表示することも可能だ。また、パナソニックのモデムカートリッジを接続すれば、内蔵されている12×12ドットの漢字ROMが、自動的に選択される。

表 3.2: 漢字 BASIC の画面モード

画面ドット数	漢字ドット数	半角文字数	設定方法
256 × 212	16 × 16	32 × 12	CALL KANJI0 : WIDTH 32
256 × 212	12 × 16	40 × 12	CALL KANJI1 : WIDTH 40
256 × 424	16 × 16	32 × 24	CALL KANJI2 : WIDTH 32
256 × 424	12 × 16	40 × 24	CALL KANJI3 : WIDTH 40
512 × 212	16 × 16	64 × 12	CALL KANJI0 : WIDTH 64
512 × 212	12 × 16	80 × 12	CALL KANJI1 : WIDTH 80
512 × 424	16 × 16	64 × 24	CALL KANJI2 : WIDTH 64
512 × 424	12 × 16	80 × 24	CALL KANJI3 : WIDTH 80

“半角文字数” は、画面に表示される半角文字の桁数と行数。当然のことながら、全角文字 (漢字) の場合は桁数が表の値の半分になるわけだ。

ここで、ひとつ注意しておいて欲しいのは、表にある行数のすべてが BASIC などで使えるわけではないこと。ファンクションキーの表示や、漢字変換のためのエリアで、画面下の 1~2 行は使われてしまう。

3.1.6 漢字テキストと漢字グラフィック

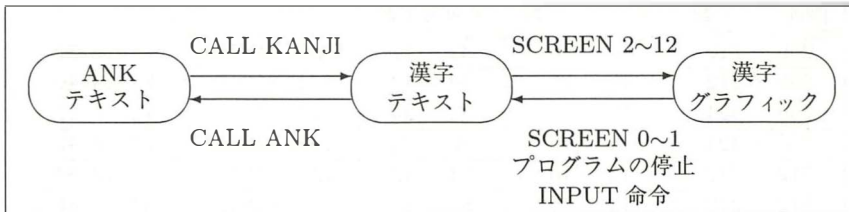
さて、画面モードには、じつはもっと複雑な問題がある。BASIC で、

```
CALL KANJI1
WIDTH 40
SCREEN 0
```

という命令を実行すると、表 3.2 の上から 2 番目の画面モードになるのはわかるかな。ここで注目して欲しいのが、“SCREEN 0” を指定しているにもかかわらず、VDP は 256 × 212 ドットの “SCREEN 5” の状態になっていること。このような状態を、**“漢字テキストモード”** という。

このモードでは、BASIC プログラムの入力や修正、INPUT 命令による入力、PRINT 文による出力などは問題なく行なえる。しかし、LINE や PAINT などのグラフィック機能は使えない。

漢字モードでグラフィックを操作するには、“SCREEN 5” などの命令で、画面を **“漢字グラフィックモード”** に切り替える必要がある。ただ、覚えておいて欲しいのは、このモードではグラフィック機能と漢字の出力は使えるけど、原則として漢字の入力はできないこと。間違えないように。以上の画面モードの切り替えを図 3.2 にまとめておいた。

図 3.2: 画面モードの切り替え

プログラムが動きはじめるとき、終了するとき、エラーが起きたときなどは、予定どおりに画面モードが設定されているか確認しよう。うっかり "CALL KANJI" を忘れていると、プログラムを入力した直後には正しく動くけど、一度電源を切ってからプログラムをロードし直すと動かない、なんて事態が発生する。

また、漢字グラフィックモードで、"LINE" 命令と "LINE INPUT" 命令を使うと、自動的に漢字テキストモードに戻ってしまう。これを避けるには、"INPUT$" や "INKEY$" 関数でキーボードを読めばいい。

3.1.7 漢字ドライバーの正しい使い方なのだ

漢字ドライバーは、MSX の漢字機能は実用にならないという過去の常識をくつがえした、"天晴 (あっぱれ)" なソフトウェア。だけど、BASIC に "あとづけ" されたために、意外な落とし穴がある。いま説明したばかりの、漢字テキストと漢字グラフィックの違いも、そんな秘孔のひとつだ。ここでは、そんな漢字ドライバーを使う上での注意点を列挙する。

MSX をリセットしてから最初に "CALL KANJI" 命令が実行されるときに、漢字ドライバーと MSX-JE を利用するためのワークエリアが用意される。そのときに BASIC の変数の内容が消滅し、"FOR〜NEXT" 命令の繰り返し回数や、"RETURN" 命令で戻る行番号を記録するための **"ソフトウェアスタック"** というワークエリアが初期化されてしまう。たとえば、

```
10 A=1
20 CALL KANJI0
30 PRINT A
```

というプログラムを実行した場合、1 回目には "0" が表示される (つまり変数 A の値が 0 になっている) けど、2 回目以降は正しく "1" が表示されるという具合。また、

```
10 GOSUB 80
```
```
70 END
80 CALL KANJI0
90 RETURN
```

のようなプログラムを実行すると、“CALL KANJI0” が実行されたとき “RETURN” 命令で戻る場所が忘れられ、プログラムの動作がおかしくなってしまうんだ。

このほかにも、漢字ドライバーがメモリー上にワークエリアを確保すると、BASIC のワークエリアがそのぶん減ってしまうという問題もある。メモリーの大きさいっぱいのプログラムや、マシン語サブルーチンを使うプログラムでは、メモリーが不足して動かなくなることもあるかもしれない。

漢字ドライバーは、プログラム内部のシフト JIS コードを、JIS コードに変換して漢字プリンターに印字する機能を持っている。この機能は、BASIC の “LPRINT” 命令と、BIOS の “LPTOUT” に対して働くものだ。しかし、プリンターを 1 ドットずつ制御して、グラフィックを印字する **“ビットイメージ印字”** では副作用が出る。そこで、ビットイメージ印字の前には、システムワークエリアの中の F418H 番地 (RAWPRT と呼ぶ) に 0 でない値を書き込んで、漢字コードの変換を禁止する必要がある。

さて、ここからの話は、上級プログラマー向けのもの。まず、表 3.3 に掲載したのが、漢字ドライバーが書き替えるフックの一覧。ほかのプログラムがこれらのフックを書き替えると、漢字ドライバーが正しく動かない可能性があるので要注意だ。

次に、フックからインタースロットコールで漢字ドライバーが呼び出されるとき

表 3.3: 漢字ドライバーが使うフック

番地	名前	機能
FDA4H	H.CHPU	画面に 1 文字を表示する
FDA9H	H.DSPC	カーソルを表示する
FDAEH	H.ERAC	カーソルを消去する
FDB3H	H.DSPF	ファンクションキーを表示する
FDB8H	H.ERAF	ファンクションキーの表示を消去する
FDBDH	H.TOTE	画面をテキストモードに切り替える
FDC2H	H.CHGE	キーボードから 1 文字を読む
FDDBH	H.PINL	BASIC のエディターが 1 行を読む
FDE5H	H.INLI	1 行を読む
FFB6H	H.LPTO	プリンターに 1 文字を書く

これらのフックをアプリケーションプログラムが使うと、漢字ドライバーが正しく動かない可能性がある。

には、裏レジスターと IX 、IY レジスターの内容が破壊される。したがって、漢字入出力に関係する BIOS について、これらのレジスターの内容が保存されると考えて作られたプログラムは、漢字モードでは正しく動かない。

第4章 V9958VDP

本章の第1節から第4節は、“V9938 MSX–VIDEO テクニカルデータブック”と“V9958仕様書”を編集部が再編集したものである。V9938とV9958の共通の機能については省略するので、“MSX–Datapack”などを参照してほしい。

本章の第5節から第7節は、“MSXマガジン”1988年12月号、1989年1月号、11月号、12月号、1990年1月号の“MSX2+テクニカル探検隊”の記事を再編集したものである。

“V9958仕様書”などのハードウェア資料では、**“VDPのモード”**が説明に使われているが、本書では、MSXマガジンの記事に合わせて、BASICの画面モードを使う。VDPのモードとBASICの画面モードは、表4.1のように対応する。

表4.1: VDPのモードとBASICの画面モード

VDPのモード	BASICの画面モード
TEXT 1	SCREEN 0 : WIDTH 40
TEXT 2	SCREEN 0 : WIDTH 80
MULTI COLOR	SCREEN 3
GRAPHIC 1	SCREEN 1
GRAPHIC 2	SCREEN 2
GRAPHIC 3	SCREEN 4
GRAPHIC 4	SCREEN 5
GRAPHIC 5	SCREEN 6
GRAPHIC 6	SCREEN 7
GRAPHIC 7	SCREEN 8 (SCREEN 10～12)

4.1 V9958 レジスター一覧

表 4.2: モードレジスター

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
R#0	0	DG	IE_2†	IE_1	M_5	M_4	M_3	0	Mode 0
R#1	0	BL	IE_0	M_1	M_2	0	SI	MAG	Mode 1
R#2	0	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	Pattern name T.B.A.
R#3	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	Color T.B.A. (Low)
R#4	0	0	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	Pattern gen. T.B.A.
R#5	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	Sprite attr. T.B.A. (Low)
R#6	0	0	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	Sprite pat. gen. T.B.A.
R#7	TC_3	TC_2	TC_1	TC_0	BD_3	BD_2	BD_1	BD_0	Text / Back drop color
R#8	MS†	LP†	TP	CB*	VR*	0	SPD	BW*	Mode 2
R#9	LN	0	S_1*	S_0*	IL	EO	$\overline{NT}$*	DC*	Mode 3
R#10	0	0	0	0	0	A_{16}	A_{15}	A_{14}	Color T.B.A. (High)
R#11	0	0	0	0	0	0	A_{16}	A_{15}	Sprite attr. T.B.A. (High)
R#12	$T2_3$	$T2_2$	$T2_1$	$T2_0$	BC_3	BC_2	BC_1	BC_0	Text / Back color
R#13	ON_3	ON_2	ON_1	ON_0	OF_3	OF_2	OF_1	OF_0	Blinking period
R#14	0	0	0	0	0	A_{16}	A_{15}	A_{14}	VRAM access base addr.
R#15	0	0	0	0	S_3	S_2	S_1	S_0	Status reg. pointer
R#16	0	0	0	0	C_3	C_2	C_1	C_0	Color palette addr.
R#17	AII	0	RS_5	RS_4	RS_3	RS_2	RS_1	RS_0	Control reg. pointer
R#18	V_3	V_2	V_1	V_0	H_3	H_2	H_1	H_0	Display adjust
R#19	IL_7	IL_6	IL_5	IL_4	IL_3	IL_2	IL_1	IL_0	Display offset
R#20*	0	0	0	0	0	0	0	0	Color burst 1
R#21*	0	0	1	1	1	0	1	1	Color burst 2
R#22*	0	0	0	0	0	1	0	1	Color burst 3
R#23	DO_7	DO_6	DO_5	DO_4	DO_3	DO_2	DO_1	DO_0	Interrupt line
R#25‡	0	CMD	VDS*	YAE	YJK	WTE	MSK	SP2	
R#26‡	0	0	HO_8	HO_7	HO_6	HO_5	HO_4	HO_3	Horizontal scroll (High)
R#27‡	0	0	0	0	0	HO_2	HO_1	HO_0	Horizontal scroll (Low)

T.B.A. : table base address
この表で "0" と書かれているビットには、かならず 0 を書き込む必要がある。
* (編集部注) ハードウェア制御用なので、普通のアプリケーションプログラムが書き替えてはいけない。
†V9938 にはあるが V9958 にはないフラグなので、かならず 0 を書き込む必要がある。
‡V9958 に新しく追加されたレジスターである。これらの初期値は 0 で、V9958 の機能が V9938 同等になる。なお、レジスター 24 は欠番である。

表 4.3: コマンドレジスター

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
R#32	SX_7	SX_6	SX_5	SX_4	SX_3	SX_2	SX_1	SX_0	Source X (Low)
R#33	0	0	0	0	0	0	0	SX_8	Source X (High)
R#34	SY_7	SY_6	SY_5	SY_4	SY_3	SY_2	SY_1	SY_0	Source Y (Low)
R#35	0	0	0	0	0	0	SY_9	SY_8	Source Y (High)
R#36	DX_7	DX_6	DX_5	DX_4	DX_3	DX_2	DX_1	DX_0	Destination X (Low)
R#37	0	0	0	0	0	0	0	DX_8	Destination X (High)
R#38	DY_7	DY_6	DY_5	DY_4	DY_3	DY_2	DY_1	DY_0	Destination Y (Low)
R#39	0	0	0	0	0	0	DY_9	DY_8	Destination Y (High)
R#40	NX_7	NX_6	NX_5	NX_4	NX_3	NX_2	NX_1	NX_0	Number of dot X (Low)
R#41	0	0	0	0	0	0	0	NX_8	Number of dot X (High)
R#42	NY_7	NY_6	NY_5	NY_4	NY_3	NY_2	NY_1	NY_0	Number of dot Y (Low)
R#43	0	0	0	0	0	0	NY_9	NY_8	Number of dot Y (High)
R#44	CH_3	CH_2	CH_1	CH_0	CL_3	CL_2	CL_1	CL_0	Color
R#45	0	MXC	MXD	MXS	DIY	DIX	EQ	MAJ	Argument
R#46	CM_3	CM_2	CM_1	CM_0	LO_3	LO_2	LO_1	LO_0	Command

表 4.4: ステータスレジスター

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
S#0	F	5SF	C	$5S_4$	$5S_3$	$5S_2$	$5S_1$	$5S_0$	Status 0
S#1	FL†	LPS†	ID_4	ID_3	ID_2	ID_1	ID_0	FH	Status 1
S#2	TR	VR	HR	BD	1	1	EO	CE	Status 2
S#3	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	Column (Low)
S#4	1	1	1	1	1	1	1	X_8	Column (High)
S#5	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	Row (Low)
S#6	1	1	1	1	1	1	EO	Y_8	Row (High)
S#7	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	Color
S#8	BX_7	BX_6	BX_5	BX_4	BX_3	BX_2	BX_1	BX_0	Border X (Low)
S#9	1	1	1	1	1	1	1	BX_8	Border X (High)

†V9938 には存在するが V9958 には存在しない機能に関するビットなので、V9958 でこれらの値は無意味である。
V9958 の ID は 00010B である。

4.2 V9958 の新機能

4.2.1 水平スクロール

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
R#25	0	CMD	VDS	YAE	YJK	WTE	*MSK*	*SP2*
R#26	0	0	HO_8	HO_7	HO_6	HO_5	HO_4	HO_3
R#27	0	0	0	0	0	HO_2	HO_1	HO_0

$HO_8 \sim HO_0$ は、画面の水平スクロール量を、SCREEN 6 と 7 では 2 ドット単位で、その他の画面モードでは 1 ドット単位で、設定する。

SP2 = 0 (初期値) ならば、水平方向画面サイズが 1 ページとなる。

SP2 = 1 ならば、水平方向画面サイズが 2 ページとなる。

MSK = 0 (初期値) ならば、画面の左端がマスクされない。

MSK = 1 ならば、SCREEN 6 と 7 では画面の左端 16 ドットが、その他の画面モードでは画面の左端 8 ドットが、マスクされ、ボーダーカラーが表示される。

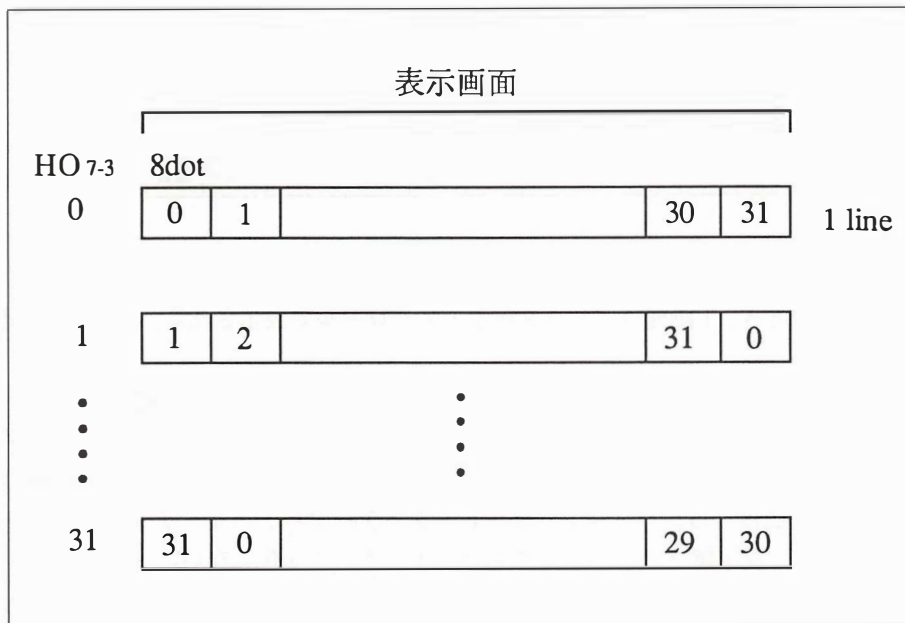
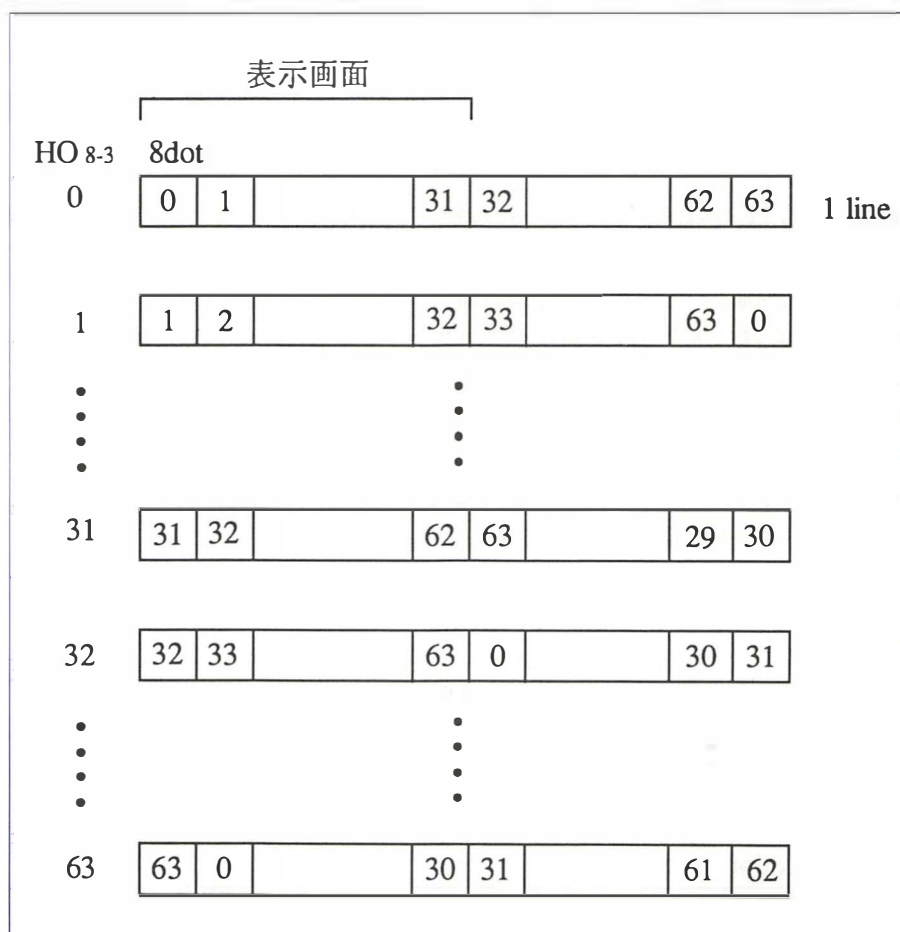
$HO_8 \sim HO_3$ に対して、表示画面は設定値だけ**左向き**に、8 ドット単位 (SCREEN 6 と 7 では 16 ドット単位) でシフトする。

図 4.1: 水平スクロール (SP2=0 の場合)

図 4.2: 水平スクロール (SP2=1 の場合)

表示画面

HO 8-3	8dot							
0	0	1		31	32		62	63
1	1	2		32	33		63	0
⋮				⋮				
31	31	32		62	63		29	30
32	32	33		63	0		30	31
⋮				⋮				
63	63	0		30	31		61	62

1 line

SP2 = 0 のとき、1 画面分のデータが水平スクロールで表示される。HO_8 は無視される。

SP2 = 1 のとき、2 画面分のデータが水平スクロールで表示される。パターンネームテーブルのベースアドレスの A_{15}には 1 を設定する。パターンネームテーブルのベースアドレスは、0~31 は、設定値の $A_{15}=0$ とした値、32~63 は、設定値の $A_{15}=1$ とした値になる。パターンジェネレーターテーブルとカラーテーブルのベースアドレスは、設定値そのままで、スクロールによって変化しない。

$HO_8 \sim HO_3$ に対して、表示画面は設定値だけ**右向き**に、1 ドット単位 (SCREEN 6 と 7 では 2 ドット単位) でシフトする。

4.2.2 ウェイト

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
R#25	0	CMD	VDS	YAE	YJK	*WTE*	MSK	SP2

WTE = 0 (初期値) ならば、ウェイト機能が無効になる。

WTE = 1 ならば、ウェイト機能が有効になる。CPUがVRAMをアクセスした際に、それによるV9958のVRAMアクセスが完了するまで、すべてのV9958ポートへのアクセスに対してウェイトがかかる。レジスターとカラーパレットへのアクセス未完および、コマンドのデータレディーによるウェイト機能はない。

4.2.3 コマンド

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
R#25	0	*CMD*	VDS	YAE	YJK	WTE	MSK	SP2

CMD = 0 (初期値) ならば、SCREEN 5~12の画面モードでのみコマンド機能が有効になる。

CMD = 1 ならば、全画面モードにおいてコマンド機能が有効になる。

SCREEN 5 ~ 12以外の画面モードでは、SCREEN 8として動作する。従ってパラメーターは、SCREEN 8のX-Y座標系で設定する。

4.2.4 YJK方式の表示

レジスターの設定

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
R#25	0	CMD	VDS	*YAE*	*YJK*	WTE	MSK	SP2

YJK = 0 (初期値) ならば、VRAM上のデータをRGB方式 (各3、3、2ビット) として扱う。スプライトの表示色は従来どおり。

YJK = 1 ならば、VRAM上のデータをYJK方式とみなし、これをRGB信号 (各5ビット) に変換し、RGB端子よりアナログ出力する。スプライトの表示色にはパレットが有効になる。

YAE は、YJK方式のデータフォーマットを選択する。

YAE = 0 の場合

アトリビュートがない。データフォーマットを次に示す。連続した4ドットをグルーピングして表わす。

C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
Y					K_L		
Y					K_H		
Y					J_L		
Y					J_H		

YAE = 1 の場合

1ドットごとにアトリビュートがある。データフォーマットを次に示す。連続した4ドットをグルーピングして表わす。

C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
Y				A	K_L		
Y				A	K_H		
Y				A	J_L		
Y				A	J_H		

A = 0 (初期値) ならば、Y、J、K は、すべて YJK 方式のデータとなる。

A = 1 ならば、Y データはカラーコードとなりカラーパレットをとおして RGB 出力される。J と K は、YJK 方式のデータとなる。

YJK 方式と RGB 方式の変換式 (参考)

$$
\begin{aligned}
R &= Y + J \\
G &= Y + K \\
B &= \tfrac{5}{4}Y - \tfrac{1}{2}J - \tfrac{1}{4}K
\end{aligned}
$$

$$
\begin{aligned}
Y &= \tfrac{1}{2}B + \tfrac{1}{4}R + \tfrac{1}{8}G \\
J &= R - Y \\
K &= G - Y
\end{aligned}
$$

(編集部注) Y の値は、アトリビュートがない場合には 0 〜 31 の整数、アトリビュートがある場合には 0 〜 30 の偶数である。J と K の値は、−32 〜 31 の整数である。YJK から RGB への変換結果は、0 〜 31 にクリッピングされる。

4.3 V9958 の廃止機能

V9938 に存在した次の機能は廃止された。

- コンポジットビデオ出力
- マウス/ライトペンインターフェース

(編集部注) MSX のマウスは、V9938 のマウスインターフェース機能を使っていないので、この機能の削除は影響ない。

4.4 V9958 ハードウェア仕様 (変更部分)

表 4.5: V9958 の端子の変更

番号	V9958			V9938	
	名称	I/O	説明	名称	I/O
4	$\overline{\text{VRESET}}$	I	HSYNC / CSYNC の3値論理の入力の分離	$\overline{\text{VDS}}$	O
5	$\overline{\text{HSYNC}}$	O	HSYNC 出力またはバーストフラグ出力	$\overline{\text{HSYNC}}$	I/O
6	$\overline{\text{CSYNC}}$	O		$\overline{\text{CSYNC}}$	I/O
8	CPUCLK / $\overline{\text{VDS}}$	O	CPUCLK 出力または VDS 出力	CPUCLK	O
21	AVDD (DAC)	I	アナログ電源	VIDEO	O
26	$\overline{\text{WAIT}}$	O	I/O WAIT 出力	$\overline{\text{LPS}}$	I
27	$\overline{\text{HRESET}}$	I	HSYNC / CSYNC の3値論理の入力の分離	$\overline{\text{LPD}}$	I

コントロールレジスター 25 ビット 5 の VDS フラグが 0 の場合に、端子 8 が CPUCLK 出力となり、VDS フラグが 1 の場合に、端子 8 が VDS 出力となる。

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
R#25	0	CMD	*VDS*	YAE	YJK	WTE	MSK	SP2

表 4.6: V9958 の直流特性

VRESET, HRESET

記号	項目	最小	標準	最大	単位
V_{IL}	低レベル入力電圧	−0.3		0.8	V
V_{IH}	高レベル入力電圧	2.2		V_{CC}	V

HSYNC, CSYNC, CPUCLK / VDS, WAIT

記号	項目	測定条件	最小	標準	最大	単位
V_{OL}	低レベル出力電圧	$I_{OL} = 1.6\text{mA}$			0.4	V
V_{OH}	高レベル出力電圧	$I_{OH} = 0.1\text{mA}$	2.4			V

G, R, B

記号	項目	測定条件	最小	標準	最大	単位
V_{RGB31}	最大出力電圧	$R_L = 470\Omega$		2.8		V
V_{RGB0}	最小出力電圧	$R_L = 470\Omega$		2.0		V
V_{P-P}	$V_{RGB31} - V_{RGB0}$	$R_L = 470\Omega$		0.8		V
D_{RGB}	V_{P-P}の偏差	$R_L = 470\Omega$			5	%

4.5 V9958 と MSX2+

MSX の画面表示を制御する部品を “Video Display Processor” 、略して VDP という。MSX2 の VDP は “V9938” というものだったけど、MSX2+以降のマシンでは “V9958” に機能アップされた。ここでは、その V9958 に追加された機能を紹介する。

4.5.1 スクリーンモードは全部で 12 種類

スクリーンモードとは、BASIC の SCREEN 命令で切り替えられる画面の状態。たとえば、横 40 桁表示で BASIC のプログラムを書きたい場合は、

```
SCREEN 0 : WIDTH 40
```

によって画面をテキスト (文字表示) モードに切り替え、グラフィックを描きたい場合は、

```
SCREEN 8
```

などで、グラフィック (図形表示) モードに切り替える。スクリーンモードが多いとなにかと面倒なので、少なくてすむならそれにこしたことはない。でも MSX2+のスクリーンモードが多いことには、それなりの理由がある。

表 4.7: MSX2+の画面モード

番号	表示方法	解像度	色数
0	文字 (1)	80 × 24 文字	文字の色と背景の色を指定する
1	文字 (2)	32 × 24 文字	文字の色と背景の色を指定する
2	テーブル (3)	256 × 192 ドット	16 色 横 8 ドットごとに 2 色
3	ビットマップ (4)	64 × 48 ドット	16 色
4	テーブル	256 × 192 ドット	16 色 横 8 ドットごとに 2 色
5	ビットマップ	256 × 212 ドット	16 色
6	ビットマップ	512 × 212 ドット	4 色
7	ビットマップ	512 × 212 ドット	16 色
8	ビットマップ	256 × 212 ドット	256 色
9	ハングル文字表示用で、日本のＭＳＸにはない。		
10	YJK / RGB	256 × 212 ドット	12,499 色 (本文参照)
11	YJK / RGB	256 × 212 ドット	12,499 色 (本文参照)
12	YJK	256 × 212 ドット	19,268 色 (本文参照)

(1) 6 ドット× 8 ドットで 1 文字を表わし、英字とカタカナを表示する。
(2) 8 ドット× 8 ドットで 1 文字を表わし、英字とカタカナとひらがなを表示する。
(3) 8 × 8 ドットのパターンの組み合わせで画面を作る。
(4) ビットマップ表示では、ドットの色を隣の色に関係なく表示可能である。

第1番目が速さの問題。グラフィックモードの画面に文字を出力すると、表示に時間がかかってしまう。だからプログラムを入力するようなときは、図形や漢字を使えない代わりに表示が速い、テキストモードが便利だ。

第2の理由は、機能が高い (表示できるドット数や色が多い) 画面ほど、たくさんのデータを必要とすること。たとえばSCREEN 8の画面データは1枚で54,272バイトもあり、これでは1枚のディスク (2DD) に12枚の絵しか記録できない。そのためROMカートリッジのゲームでは、色数に制限があるかわりにデータが少なくて動作が速い、SCREEN 2を使ってメモリーを節約することが多くなっている。

第3に、互換性を保ちながら機能を追加したために、多くのスクリーンモードが必要になったこと。最初のMSXでは、SCREEN 0から3までの4種類のスクリーンモードしかなかった。それがMSX2になり、高解像度グラフィック表示のためSCREEN 4から8が追加。さらにMSX2+では、色数を増やすためにSCREEN 10から12が追加されたというわけ。

これらのスクリーンモードをまとめたのが表4.7。でも実際にはスクリーンモードの問題はこれで終わりではなく、縦方向の解像度を2倍に増やすインターレースモードや、MSX2+で新しく追加された漢字モードなどもある。

4.5.2 VDPのレジスターをコントロールする

MSXの画面表示を制御するVDPの中には、マシンの心臓部であるCPUと同様に“レジスター”というものがある。そしてこのVDPのレジスターは、I/Oポートをとおすことで CPUが操作できるものだ。中でも、CPUがVDPを制御するために使うレジスターを **“コントロールレジスター”**、CPUがVDPの状態を知るために使うレジスターを **“ステータスレジスター”** と呼んでいる。このほかにも、VDPに高度な命令を実行させるためにCPUが操作する **“コマンドレジスター”** があるけど、本書のプログラムでは使用しないので、説明は省略する。

CPUとVDPを接続するI/Oポートの番地は、普通は98Hから9BHまでを使っている。でも正確には表4.8に掲載したように、ROMの6番地と7番地の内容で

表4.8: VDPのI/Oポート

ポート名	R/W	I/O番地	用途
ポート0	READ	ROMの6番地の内容	VRAM読み出し
ポート1	READ	ROMの6番地の内容+1	ステータスレジスター
ポート0	WRITE	ROMの7番地の内容	VRAM書き込み
ポート1	WRITE	ROMの7番地の内容+1	コントロールレジスター
ポート2	WRITE	ROMの7番地の内容+2	パレットレジスター
ポート3	WRITE	ROMの7番地の内容+3	間接指定レジスター

決まってくる。これは、MSX 本体の外に VDP を増設できるようにとの配慮から。なお、同じポート 0 でも、書き込む場合と読み出す場合とでは、I/O ポートの番地が異なることがあるので注意しよう。

さて、VDP のコントロールレジスターの値を設定するためには、まず、表 4.8 のポート 1 に設定したいデータを書き込み、続けて同じポート 1 に“レジスター番号+128”を書き込む。この“続けて”という条件は意外と重要で、2 回の書き込みの間に割り込みが起こると VDP が混乱してしまうんだ。だからこの場合は、まず“DI”命令で割り込みを禁止しておいてから、ふたつのデータを続けて書き込む方法が一般的に使われている。

ところで、コントロールレジスターというのは書き込み専用のもの。一度設定された値は、読み出すことができない。だから、レジスターの特定のビットだけを書き替えたい、なんて場合は都合が悪くなってしまう。そこで通常は、コントロールレジスターに書き込む値を、表 4.9 のようなシステムワークエリア (RAM) にも書き込んでおくという方法をとる。たとえば、コントロールレジスター 1 のビット 4 を 1 に変えたい場合には、RAM の F3E0H 番地の内容を読んでビット 4 を 1 に変え、その値をコントロールレジスター 1 と RAM の F3E0H 番地に書き込むという具合だ。あとで紹介するリスト 4.9 の走査線割り込みのサンプルプログラムの中では、“WRTVDP”というサブルーチンがこれと同じ動作を行なっている。

次に、ステータスレジスターの値を読むための方法を説明する。これは、コントロールレジスター 15 に読みたいステータスレジスターの番号を設定し、VDP のポート 1 の値を読み、コントロールレジスター 15 を 0 に戻すという手続きを、割り込みを禁止したままで行なうというもの。リスト 4.9 のサンプルプログラムの中では、“_VDPSTA”というサブルーチンがこれにあたる。

表 4.9: コントロールレジスターの保存場所

レジスター番号	VDP 関数番号	保存番地	ラベル
0	0	F3DFH	RG0SAV
⋮	⋮	⋮	⋮
7	7	F3E6H	RG7SAV
8	9	FFE7H	RG8SAV
⋮	⋮	⋮	⋮
23	24	FFF6H	RG23SA
25	26	FFFAH	RG25SA
26	27	FFFBH	RG26SA
27	28	FFFCH	RG27SA

表 4.10: そのほかの便利なシステムワークエリア

番地	ラベル	意味
F341H	RAMAD0	ページ0のRAMのスロット番号*
F342H	RAMAD1	ページ1のRAMのスロット番号*
F343H	RAMAD2	ページ2のRAMのスロット番号*
F344H	RAMAD3	ページ3のRAMのスロット番号*
FAF5H	DPPAGE	ディスプレーページ番号
FAF6H	ACPAGE	アクティブページ番号
FD9AH	H.KEYI	割り込みフック
FD9FH	H.TIMI	タイマー割り込みフック

*** ディスクがある場合のみ有効。**

表 4.11: MSX2+に追加、変更されたシステムワークエリア

番地	ラベル	意味
0FAFCH	MODE	次表参照
0FAFDH	NORUSE	漢字ドライバーのワークエリア
0FD0AH	SLTWRK+1	漢字ドライバーのワークエリア
⋮	⋮	
0FD0FH	SLTWRK+6	
0FFFAH	RG25SA	VDPレジスターのセーブ
0FFFBH	RG26SA	
0FFFCH	RG27SA	

表 4.12: 0FAFCH番地 (MODE) の詳細

ビット	意味
b_7	1ならばカタカナ、0ならばひらがな
b_6	1ならば第2水準漢字ROMあり
b_5	1ならばSCREEN 11 、0ならばSCREEN 10
b_4	内部で使用
b_3	1ならばSCREEN 0~3でVRAM番地に3FFFHをマスク
b_2	VRAM容量
b_1	00 : 16KB 、01 : 64KB 、10 : 128KB
b_0	1ならばローマ字カナ変換

もっとも、MSX2や2+のROMにはこれらのサブルーチンと同じ機能のBIOSがあるので、普通は自分でサブルーチンを作らずにBIOSを使えばいい。でも、これらのBIOSはサブROMにあったり、処理中にサブROMを呼び出したりするので、

多少時間がかかるという難点がある。そのため、走査線割り込みのような処理には都合が悪いので、あえて BIOS を使わないわけだ。

表 4.10 ～ 4.12 に、そのほかのシステムワークエリアをまとめておいたので、参考にしてほしい。

4.5.3 V9958 のレジスター

図 4.3 に掲載したのは、V9958 に追加された 3 個のコントロールレジスター。これらのレジスターは書き込み専用のもので、書き込まれた値を表 4.9 のシステムワークエリアに記録する。コントロールレジスター 24 がないことや、レジスター番号と BASIC の VDP 関数の番号が異なることに注意しよう。

V9958 で追加された機能の大部分は、コントロールレジスター 25 で制御される。有名な YJK(自然画) 表示と横スクロールは後回しにして、残りの機能から紹介するぞ。

レジスター 25 のビット 7 には、かならず 0 を書き込もう。ビット 5 は “VDS” と呼ばれ、VDP の端子 8 の機能を制御する。けれども、普通のプログラムがこのビットを書き替えることは禁止されている。

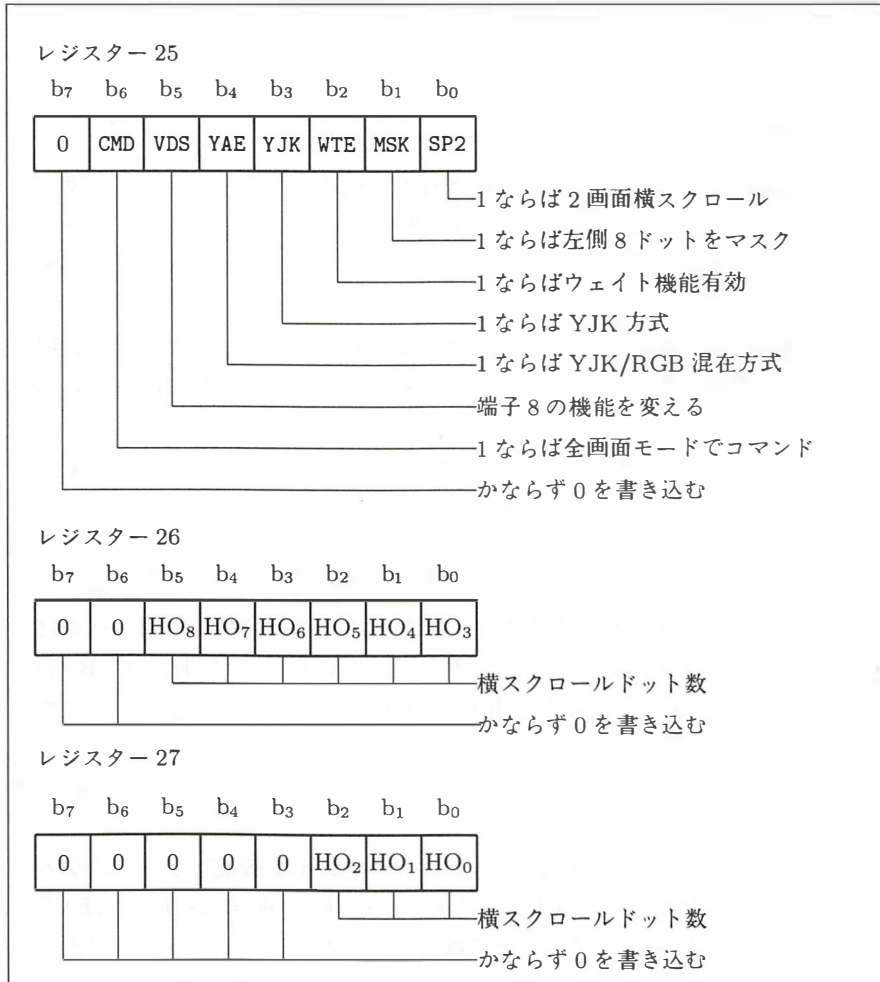
またビット 6 に 0 を書き込めば、V9938 と同様に SCREEN 5 から 8 の画面に対してのみ **“VDP コマンド”** が使えるようになる。これとは逆に 1 を書き込めば、全画面モードで VDP コマンドが使えるわけだ。この VDP コマンドとは、BASIC の COPY 命令や LINE 命令のような仕事を、VDP にさせる機能。細かい話になるけど、SCREEN 5 から 8 以外の画面に対する VDP コマンドでは、128 キロバイトのビデオ RAM の中の場所を、SCREEN 8 のように X 座標が 0 から 255 、Y 座標が 0 から 511 の値で指定する。

さらにビット 2 に 1 を書き込むと、VDP の **“ウェイト機能”** が有効になる。これは CPU がビデオ RAM を読み書きするとき、CPU の動作が速すぎれば VDP が CPU に “WAIT 信号” を送って待たせる機能だ。ただし、パレットレジスターへの書き込みと、VDP コマンドによる転送には、ウェイト機能がない。

4.5.4 VDP による横スクロール

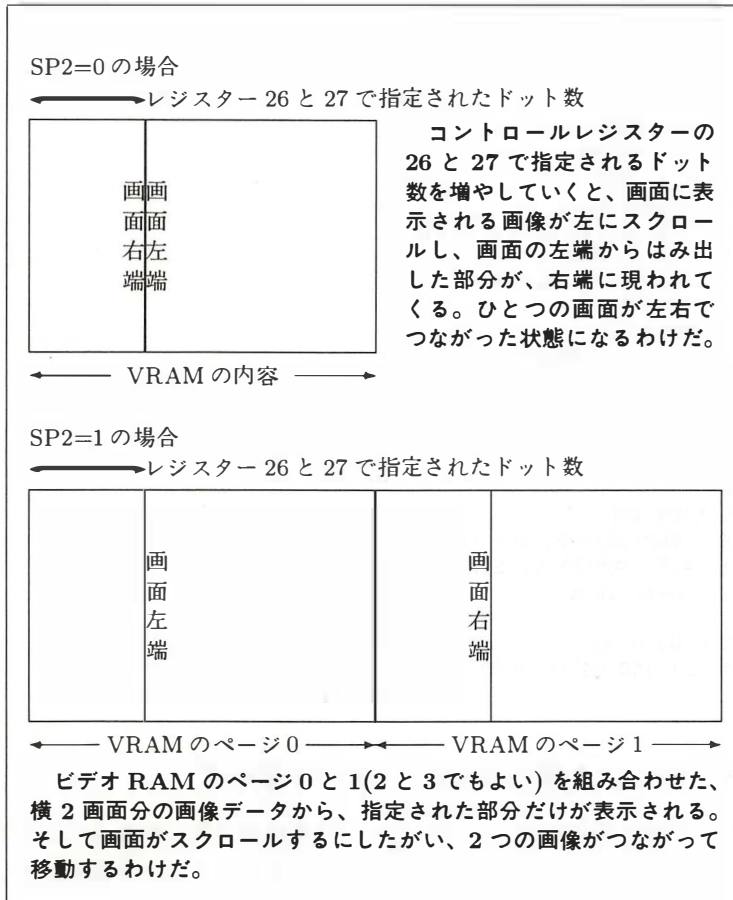
横スクロールには、1 画面分の画像データを使う方法と、2 画面分の画像データを使う方法がある。それぞれの場合について、順番に説明していこう。

まず、コントロールレジスター 25 のビット 0 “SP2” が 0 の場合には、図 4.4 の上のように 1 画面分のデータによる横スクロールが起こる。スクロールのドット数は、レジスター 26 と 27 で指定することができる。レジスター 26 に 0 から 63 の値を書き込むと、その値× 8 ドット単位で画面が左にスクロールし、レジスター 27 に

図 4.3: V9958に追加されたコントロールレジスターの機能一覧

0から7の値を書くとその数だけ右に移動する。ただしSCREEN 6と7では、指定した数の2倍のドット数のスクロールが起こるから注意しよう。ドット数を0から255まで順番に増やしていくと、画面が左へスクロールし、はみだした部分が右端から現われてくる。

一方レジスター25のビット0が1の場合には、図4.4の下のように、2画面分の画像データから指定された部分が表示され、横スクロールがはじまる。

このときの画像データは、ビデオRAMのページ0と1または2と3に記憶され

図 4.4: 2 種類の横スクロールの仕組み

ている。ページが 0 と 1 の場合はディスプレーページを 1 に、2 と 3 の場合にはディスプレーページを 3 に設定しよう。横スクロールのドット数は 0 から 511(つまり 1 画面スクロールの 2 倍) になる。

また、レジスター 25 のビット 1 を 1 にすると、画面左端の 8 ドット (SCREEN6 と 7 では 16 ドット) が表示されず、代わりにその場所に画面の周辺色が表示される。これは横スクロール、とくに 1 画面分のデータによる横スクロールをする場合、画面からはみ出した部分がすぐに反対側から現われるのを隠すのに便利だ。リスト 4.1 に横スクロールのプログラム例を掲載しておくので、参考にしよう。

リスト 4.1 (HSCROLL.BAS)

```
100 ' hscroll.bas
110 ' by nao-i on 26. Oct. 1989
120 '
130 ONSTOP GOSUB 290: STOP ON
140 DEFINT A-Z: SCREEN 5
150 COLOR 15,1,1: CLS
160 LINE (0,0)-(255,211)
170 SET PAGE 1,1: COLOR 15,8,1: CLS
180 LINE (0,0)-(255,211)
190 V6=VDP(26) :VDP(26)=V6 OR 3
200 '*** scroll
210 FOR V7=0 TO 63
220   GOSUB 320: VDP(27)=V7: VDP(28)=7
230   FOR V8=6 TO 0 STEP -1
240     GOSUB 320: VDP(28)=V8
250   NEXT V8
260 NEXT V7
270 GOTO 200
280 '*** restore VDP
290 STOP OFF: VDP(26)=V6: SET PAGE 0,0
300 COLOR 15,4,7: SCREEN 0: END
310 '*** wait next vsync
320 TIME=0
330 IF TIME=0 GOTO 330
340 IF (VDP(-2) AND 64)=0 GOTO 340
350 RETURN
```

4.5.5 何があっても裏技は使ってはいけないぞ

VDPのレジスターの中には、かならず0を書き込めとか、かならず1を書き込めとか指定されているビットがある。この指定を無視して変な値を書き込むようなことは、何が起こるかわからないので、絶対にやってはいけない。

また、YJK=0とYAE=1の組み合わせのように、VDPの動作が仕様書で決められていない設定があるけど、こうした仕様書に書かれてない“裏技”も、使ってはいけない。たとえ自分が持っているマシンで問題なく動いたとしても、それはたまたま動いただけのこと。ほかのMSXマシンで正常に動作するという保障はどこにもない。また、現在のV9958に対しては有効な裏技であっても、今後VDPが改良されたり、ヤマハ以外のメーカーがV9958互換のVDPを作ったりすれば(いまのところV9958はすべてヤマハ製)、同じ裏技が有効とは限らない。

これと同じように、CPUの裏技(正式には**“未定義命令”**という)も一切使ってはいけない。過去の例でも、Z80の裏技を使ったために、ビクターのHC-95のターボモード(Z80の上位互換CPUであるHD64180を使っている)で暴走したソフトウエアがあった。

4.6 YJK 方式を解剖する

4.6.1 テレビ放送と YJK 方式

MSX2+の自然画モードには、従来の RGB 方式に代わって YJK 方式が使われている。この YJK 方式は、カラーテレビ放送の信号に似ている。

この本の読者の多くは知らないかもしれないけれど、昔のテレビ放送は白黒で、当然すべてのテレビ受信機は白黒受信機だった。その後カラー放送をはじめるときに、ふたつのことが問題になった。カラー画面を RGB(赤緑青) の 3 色に分解して放送すると、3 つのチャンネルをつかってしまう。そして、白黒受信機でもカラー放送を受信できる必要がある。

そこで、人間の目の性質が利用された。網膜は、明かるさを感じる細胞と色を感じる細胞を持っている。明かるさを感じる細胞の数が多く、色を感じる細胞の数が少ないので、人間の目は色の変化に鈍感なのだ。

カラーテレビ放送では、**"輝度"** を表わす Y 信号と **"色相"** を表わす UV 信号の組み合わせで、色が表わされている。人の目が色の変化に鈍感なことを利用すると、UV 信号の量 (専門的には周波数帯域という) は少しですみ、ひとつのチャンネルで YUV 信号をまとめて送れる。また、輝度を表わす Y 信号は白黒放送の信号と同じなので、白黒受信機でカラー放送を見ると、Y 信号だけが表示され、正常な白黒画面に見えるわけだ。

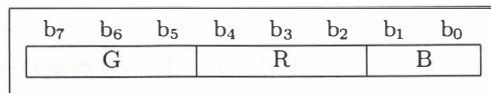
このようにテレビでは、電波の量 (つまりチャンネル数) を増やさずにカラー画面を放送するために、YUV 方式が使われた。一方、MSX2+では、ビデオ RAM を増やさずに色数を増やすために、YUV 方式に似た YJK 方式が使われる。

4.6.2 RGB 方式と YJK 方式のデータ構造

RGB 方式 (SCREEN 8)

SCREEN 8 では、R、G、B の明かるさをそれぞれ 3、3、2 ビットで表わし、1 ドットごとに 256 色中の任意の色を表示できる。

図 4.5: RGB 方式画面のデータ構造

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
G			R			B	

YJK 方式 (SCREEN 12)

SCREEN 12 では、横 4 ドットを 1 組みとする YJK 方式が使われる。図 4.6 の Y_0から Y_3までは、各ドットの明かるさを 5 ビット (つまり 32 段階) で、K と J は 4 ドット全体の色相を 12 ビット (4096 色) で表わす。YJK 方式のデータを RGB 方式に変換する方法は次の通りだ。

$$
\begin{aligned}
R &= Y + J \\
G &= Y + K \\
B &= 1.25Y - 0.5J - 0.25K
\end{aligned}
$$

ただし、Y の値は 0 から 31 まで、J と K の値は−32 から 31 までだ。上の式で計算した R 、G 、B の値が 0 よりも小さくなれば、その値の代わりに 0 が出力され、同様に 31 よりも大きくなれば、31 が出力される。このような処理を、“0 から 31 に**クリッピング**する” という。

たとえば、次にあげた 4 バイトのデータは、$Y = 0$ 、$J = K = -32$ で黒を表わすことがわかるかな。電卓片手に計算してみよう。

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0

これとは逆に、RGB のデータを YJK のデータへ変換することを考えてみよう。つまり、RGB への変換式を 3 元連立 1 次方程式とみなし、これを Y 、J 、K についてそれぞれ解けばいいわけだ。高校の “代数・幾何” レベルの問題だぞ。答えは次にあげた 3 つの式。ちゃんとできたかな。

$$
\begin{aligned}
Y &= (2R + G + 4B)/8 \\
J &= (6R - G - 4B)/8 \\
K &= (-2R + 7G - 4B)/8
\end{aligned}
$$

YJK 方式の画像を表示するときにも、VDP から出力される信号は前に説明した式で従来と同じアナログ RGB 信号と、コンポジット (ビデオ) 信号に変換されているので、MSX2+に特別なモニターテレビは必要ない。

YJK 方式の画面では、SCREEN 8 に似て、基本的には 1 バイトが 1 ドットに対応する。しかし、J と K の値は横 4 ドットごとに指定されるので、たとえば、

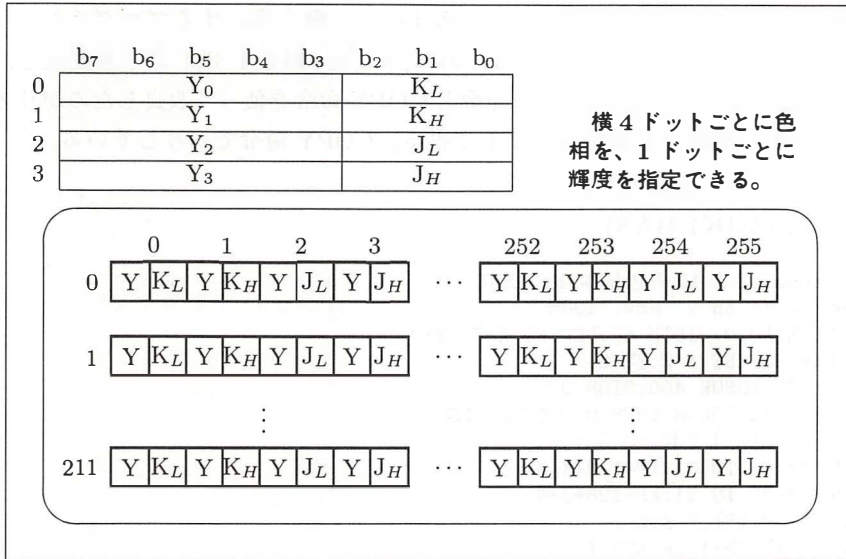
```
PSET (0,0),0
```

を行なうと、(0,0) から (3,0) までの、4 ドットの K の値が変わってしまう。そのため SCREEN 12 で LINE 命令などを使うと、“色化け” が起きるわけだ。これに関しては、あとでもう一度説明する。

図 4.6: YJK 方式画面のデータ構造

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	Y_0					K_L		
1	Y_1					K_H		
2	Y_2					J_L		
3	Y_3					J_H		

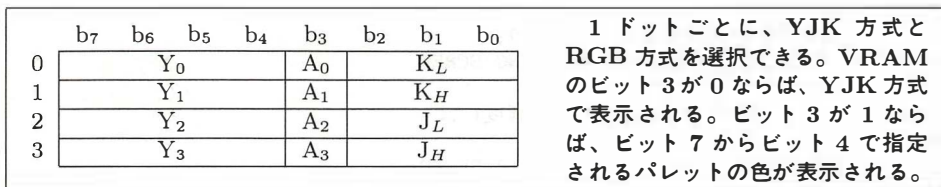
横 4 ドットごとに色相を、1 ドットごとに輝度を指定できる。

	0		1		2		3			252		253		254		255	
0	Y	K_L	Y	K_H	Y	J_L	Y	J_H	···	Y	K_L	Y	K_H	Y	J_L	Y	J_H
1	Y	K_L	Y	K_H	Y	J_L	Y	J_H	···	Y	K_L	Y	K_H	Y	J_L	Y	J_H
⋮																	
211	Y	K_L	Y	K_H	Y	J_L	Y	J_H	···	Y	K_L	Y	K_H	Y	J_L	Y	J_H

混在方式 (SCREEN 10、11)

YJK 方式の欠点は、横 4 ドットごとにしか色を指定できないので、自然画に文字や線画を重ね書きしにくいことだ。そこで、RGB 方式の SCREEN 5 と YJK 方式の SCREEN 12 の長所を合わせ持つ SCREEN 10 と 11 が用意された。これらの画面モードでは、自然画に文字を重ねて表示することが容易だが、色数は SCREEN 12 よりも少なくなる。

図 4.7: 混在方式画面のデータ構造

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	Y_0				A_0	K_L		
1	Y_1				A_1	K_H		
2	Y_2				A_2	J_L		
3	Y_3				A_3	J_H		

1 ドットごとに、YJK 方式と RGB 方式を選択できる。VRAM のビット 3 が 0 ならば、YJK 方式で表示される。ビット 3 が 1 ならば、ビット 7 からビット 4 で指定されるパレットの色が表示される。

4.6.3 色見本のプログラム

MSX2+の最大のウリは、19,268 色表示の SCREEN 12。ビデオ RAM を増やさずに (MSX2 と同じ 128 キロバイト) 色数を増やすため、YJK という表示方式が取

られている。これまで説明してきたように、輝度(明かるさ)を表わすYの値と、色相を表わすJとKの組み合わせで、色を指定する方式だ。

リスト4.2は、そのYJKで表現できるすべての色を表示するプログラム。ただし、ここではJとKの値を設定するために、32,768回もPSET命令を使用し、時間がかかってしまう。そこでLINE命令とCOPY命令を使って改良したのがリスト4.3。Kの値を左側の1列のみPSETで書き、COPY命令で複写している。

リスト4.2 (YJK1.BAS)

```
100 'screen 11と12の色見本 by PSET
110 'by nao-i on 2. Nov. 1988
120 CALL KANJI0:WIDTH 64:DEFINT A-Z:YY=0
130 CLEAR:DEFINT A-Z:YY=0
140 ON STOP GOSUB 400:STOP ON
150 SCREEN 12:COLOR &HF8,0,0:CALL CLS
160 'ＶＲＡＭのＪとＫを設定する
170 FOR K=-32 TO 31:YP=112+K+K
180 FOR J=-32 TO 31:XP=128+J*4
190 PSET (XP,YP),K AND 7
200 PSET (XP,YP+1),K AND 7
210 PSET (XP+1,YP),(K AND 56)¥8
220 PSET (XP+1,YP+1),(K AND 56)¥8
230 PSET (XP+2,YP),J AND 7
240 PSET (XP+2,YP+1),J AND 7
250 PSET (XP+3,YP),(J AND 56)¥8
260 PSET (XP+3,YP+1),(J AND 56)¥8
270 NEXT:NEXT
280 'Ｙの値を変化させる
290 SC=12:NS=1:SCREEN 12
300 FOR Y=1 TO 31:GOSUB 360:NEXT
310 FOR Y=30 TO 0 STEP -1:GOSUB 360:NEXT
320 SC=11:NS=1:SCREEN 11
330 FOR Y=2 TO 30 STEP 2:GOSUB 360:NEXT
340 FOR Y=28 TO 0 STEP -2:GOSUB 360:NEXT
350 GOTO 280
360 'ＶＲＡＭのＹを書き替えるサブルーチン
370 IF NS THEN LOCATE 1,0:PRINT USING"SCREE
N ##";SC:NS=0
380 LOCATE 1,1:PRINT USING"Ｙの値（輝度）は
##です。";Y
390 LINE(0,48)-(255,175),(Y XOR YY)*8,BF,XO
R:YY=Y:RETURN
400 'on stop で呼び出される
410 SCREEN 0:COLOR 15,4,7:END
```

リスト 4.3 (YJK2.BAS)

```
100 'screen 11と12の色見本
110 'by nao-i on 2. Nov. 1988
120 CALL KANJI0:WIDTH 64:DEFINT A-Z:YY=0
130 CLEAR:DEFINT A-Z:YY=0
140 ON STOP GOSUB 380:STOP ON
150 SCREEN 12:COLOR &HF8,0,0:CALL CLS
160 'ＶＲＡＭのＪとＫを設定する
170 FOR K=-32 TO 31:YP=112+K+K
180 PSET (0,YP),K AND 7
190 PSET (0,YP+1),K AND 7
200 PSET (1,YP),(K AND 56)¥8
210 PSET (1,YP+1),(K AND 56)¥8:NEXT
220 FOR J=-32 TO 31:XP=128+J*4
230 LINE(XP+2,48)-(XP+2,175),J AND 7
240 LINE(XP+3,48)-(XP+3,175),(J AND 56)¥8
250 COPY(0,48)-(1,175)TO(XP,Y0):NEXT
260 'Ｙの値を変化させる
270 SC=12:NS=1:SCREEN 12
280 FOR Y=1 TO 31:GOSUB 340:NEXT
290 FOR Y=30 TO 0 STEP -1:GOSUB 340:NEXT
300 SC=11:NS=1:SCREEN 11
310 FOR Y=2 TO 30 STEP 2:GOSUB 340:NEXT
320 FOR Y=28 TO 0 STEP -2:GOSUB 340:NEXT
330 GOTO 260
340 'ＶＲＡＭのＹを書き替えるサブルーチン
350 IF NS THEN LOCATE 1,0:PRINT USING"SCREE
N ##";SC:NS=0
360 LOCATE 1,1:PRINT USING"Ｙの値（輝度）は
##です。";Y
370 LINE(0,48)-(255,175),(Y XOR YY)*8,BF,XO
R:YY=Y:RETURN
380 'on stop で呼び出される
390 SCREEN 0:COLOR 15,4,7:END
```

4.6.4 必殺のロジカルオペレーションなのだ

ロジカルオペレーションとは日本語で論理演算のこと。2進数の1ビットごとに行なわれる計算を意味する。これにはAND、OR、XOR、NOTの4種類がある。

AND演算とは、2つの値の両方が1になっているビットを1にする演算。たとえば、

```
X%=&B00000011
Y%=&B00000101
```

に対してAND演算を行なうと、変数X%と変数Y%のビット0(一番右側のビット)のみが1なので、

```
X% AND Y%=&B00000001
```

となるわけだ。同様にして、OR演算では2つの値の両方もしくは一方が1になっているビットを1に、XOR演算では2つの値の一方だけ1になっているビットを1にする。またNOTとは、1つの値の各ビットを反転させる演算で、画像データを反転して書き込むことはPRESET演算とも呼ばれる。言葉で説明するとわかりにくいので、表4.13に内容をまとめてみた。

表4.13: ロジカルオペレーション

記号	意味	例
PSET	指定された色をそのまま書き込む	
AND	元の色との論理積を書き込む	0011 AND 0101 → 0001
OR	元の色との論理和を書き込む	0011 OR 0101 → 0111
XOR	元の色との排他的論理和を書き込む	0011 XOR 0101 → 0110
PRESET	指定された色のビット反転を書き込む	NOT 0101 → 1010

さて、MSX2やMSX2+でビデオRAMを扱うには、こうしたロジカルオペレーションの考えが必要になる。例題として、前に掲載したリスト4.3のプログラムで、ビデオRAMのYの値を1(2進数で00001)から2(00010)へ増やすことを考えてみよう。つまり、

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
0 0 0 0 1 ? ? ?

というビデオRAMの内容を、

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
0 0 0 1 0 ? ? ?

に変えるわけだ。ビット2から0にはJまたはKの値が入っているので、ここを変えてはいけない。

筆者が考えたのは、元のYの値の1と目的の値の2のXORである3を8倍し、

```
LINE(0,48)-(255,175), 24, XOR
```

を行なうこと。1回の書き替えでのY値が1から2に変わるわけだ。これがリスト4.3の370行、

```
LINE (0,Y0)-(255,Y0+127), (Y XOR YY)*8, BF, XOR
```

の意味。Yは目的のYの値、YYは元のYの値を表わしている。

LINE命令でロジカルオペレーション(論理演算)を指定すると、書き込もうとする色と元の色との間で演算を行なった結果が、ビデオRAMに書き込まれる。たとえばSCREEN 12の画面に対して、

```
LINE (0,0)-(255,211), &B11111000, BF, AND
```

を行なうと、ビデオRAMのビット7からビット3まではそのままで、ビット2からビット1までが0になり、画面全体が白黒の濃淡で表示される。

4.6.5 いわゆる色化け

YJK画面のデータ構造は、前に図4.6にまとめたとおり。SCREEN 8と同じように、基本的には画面の1ドットがビデオRAMの1バイトに対応。横256×縦212ドットの画面を、54,272バイトのビデオRAMで表示している。でも、色相を指定するJとKの値は横4ドットごとに指定されるので、たとえばSCREEN 12で、

```
PSET (3,0),3
```

を実行すると、(3,0)の1ドットだけでなく(0,0)から(3,0)までの4ドットのJの値が変わり、この部分が赤くなってしまう。SCREEN 12の画面を“sample.s12”というファイルにセーブしてから、リスト4.4を実行すると、いわゆる“色化け”が起こってしまうはず。実際に試してみよう。原則としてSCREEN 12では文字や線を表示できないのだ。

リスト4.4 (S12.BAS)

```
10 ' 色化けの例
20 CALL KANJI
30 SCREEN 12:COLOR &HF9,2
40 BLOAD "sample.s12",S
50 LINE (0,0)-(211,211),3
60 LOCATE 4,6:COLOR &HF9,2
70 PRINT "SCREEN 12 では色が化ける。"
80 GOTO 80
```

4.6.6 SCREEN 10と11は何がどう違うのか

SCREEN 10と11のデータ構造は、前の図4.7にまとめたとおり。どちらも機能的にはまったく同じだ。BLOADされた画面を表示する場合にも、これらのスクリーンモードの違いは表われない。それでは何が違うのか。BASICの命令などで、画面に図形や文字を書き込むとき、問題になってくる。

リスト4.5が、その違いを見せるプログラム例。YJK方式で記録された画面データをBLOADし、

```
CIRCLE (128,106),100,6
```

を行なってみよう。SCREEN 10では、ビデオRAMのビット3が1になり、ビット7からビット4に6(サークル命令で指定した赤の色番号。2進数では0110)が書き込まれる。そのため背景のYJK画面を壊さずに、赤い円を描くことができる。しかしSCREEN 11では、ビデオRAMに直接6(つまり8ビットに渡って00000110)が書かれるので、“色化け”が起こってしまう。

この例でもわかるように、YJK画面に文字や図形を重ねて描くにはSCREEN 10を。YJKの画面データを加工するにはSCREEN 11を使う必要がある。

リスト 4.5 (S10.BAS)

```
100 ' 色化けを回避するには
110 SCREEN 12
120 BLOAD "sample.s12",S
130 SCREEN 10
140 CIRCLE (128,106),100,6
150 FOR I%=1 TO 50:BEEP:NEXT
160 SCREEN 12
170 BLOAD "sample.s12",S
180 SCREEN 11
190 CIRCLE (128,106),100,6
200 GOTO 200
```

4.6.7 SCREEN 11でもテロップを使うには

リスト4.6は、SCREEN 11の画面に文字を書き込むプログラム。文字を表示するにはSCREEN 10を使うと書いたばかりだけど、PRINT命令の関係でSCREEN 11の方が都合がいいこともある。

たとえば、SCREEN 10でLINEやPUT KANJI命令を使うと、YJK画面にはRGB方式で線や文字が書き込まれる。しかしPRINT命令を使うと背景に正方形の枠が出現し、その中に文字が書かれてしまう。これではどうも、テロップとして使うには適さない。

そこで登場するのがSCREEN 11。SET PAGE命令でビデオRAMをページ1に切り替え、そこに背景の画面をBLOADする。そしてページを0に戻してから、前景色を7、背景色を0に切り替え、PRINT命令で文字を表示。さらにこの文字を、COPY命令で **“TAND”** を指定して、ページ1の画像データの目的の部分に複写するというわけ。

TANDとは、色番号が0(透明)の部分は複写せず、ほかの色の部分だけをAND演算しながら複写する機能。その結果、色番号が0で書かれた文字の枠は無視され、7(水色)で書かれた文字だけが複写される。文字を表示したい部分のビデオRAMのビット7からビット3が0になるわけだ。

次に、同じ文字を (C×16+8) で指定された色でページ 0 に書く。この C の値は、プログラムのはじめの方で入力した色番号。

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
C				1	0	0	0

というように、ビット 7 からビット 4 が目的の色番号で、RGB 表示を指定するビット 3 が 1 、残りのビット 2 からビット 0 が 0 となる。この文字を、今度は TOR を指定して複写すると、文字の部分のビデオ RAM の内容は、

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
C				1	元の値		

となり、背景の YJK 画面を壊さずに、RGB 方式で文字を表示できるというわけだ。このように、べつの場所に書いた文字を COPY 命令で複写すると、PRINT 命令にロジカルオペレーション機能がないという欠点を補える。

MSX2+の YJK 方式は難しいけど、うまい使い方をすれば、すばらしい効果を得られるモード。みんな頑張って研究してみよう。

リスト 4.6 (S11.BAS)

```
100 ' SCREEN 11 のテロップ
110 ' by nao-i on 4. Nov. 1988
120 SCREEN 0:WIDTH 32:CALL KANJI0
130 DEFINT A-Z:ON STOP GOSUB300:STOP ON
140 FILES:PRINT:INPUT "file name";FF$
150 OPEN FF$ FOR INPUT AS #1:CLOSE #1
160 INPUT "string";SS$
170 INPUT "color(1...15)";C
180 INPUT "X(0...240)";X
190 INPUT "Y(0...196)";Y
200 SCREEN 12:SET PAGE 1,1:BLOAD FF$,S
210 SCREEN 11
220 SET PAGE 0,0:COLOR 7,0:CALL CLS
230 L=LEN(SS$)*8-1
240 LOCATE 0,0:PRINT SS$
250 COPY (0,0)-(L,15),0 TO (X,Y),1,TAND
260 LOCATE 0,0:COLOR C*16+8,0:PRINT SS$
270 COPY (0,0)-(L,15),0 TO (X,Y),1,TOR
280 SET PAGE 1,1
290 GOTO 290
300 '*** called by STOP ***
310 SET PAGE 0,0:COLOR 15,4,7
```

4.6.8 SCREEN 12で文字表示をするための裏技だ

完全YJK方式のSCREEN 12には、原則として文字や線を表示することはできない。けれどもロジカルオペレーションを活用して、白い文字を表示する裏技があるので紹介しよう。それがリスト4.7のプログラム。プログラムの構造自体は、さきほどのリスト4.6とほとんど同じ。でも注意してほしいのが、

```
COLOR &HF8, 0
```

と、

```
COPY  …  , TOR
```

の2ヵ所の部分だ。これで、文字を表示する部分のYの値を最大の31に設定でき、背景より白っぽい色で文字を表示できるようになる。逆に文字の色を3に、ロジカルオペレーションをTANDにすれば、黒っぽい文字が表示される。

プログラムを実行すると、まずディスクのファイル一覧が表示されるので、背景に使いたい画像ファイルを指定しよう。続いて画面に表示するテロップを書き、位置を座標で入力する。これでSCREEN 12の画面に、テロップが表示されるはずだ。プログラムを修正して、テロップをいっぱい出すのもおもしろいかも。

リスト4.7 (S12T.BAS)

```
100 ' SCREEN 12 のテロップ
110 ' by nao-i on 4. Nov. 1988
120 SCREEN 0:WIDTH 32:CALL KANJI0
130 DEFINT A-Z:ON STOP GOSUB260:STOP ON
140 FILES:PRINT:INPUT "file name";FF$
150 OPEN FF$ FOR INPUT AS #1:CLOSE #1
160 INPUT "string";SS$
170 INPUT "X(0...240)";X
180 INPUT "Y(0...196)";Y
190 SCREEN 12:SET PAGE 1,1:BLOAD FF$,S
200 SET PAGE 0,0:COLOR &HF8,0:CALL CLS
210 L=LEN(SS$)*8-1
220 LOCATE 0,0:PRINT SS$
230 COPY (0,0)-(L,15),0 TO (X,Y),1,TOR
240 SET PAGE 1,1
250 GOTO 250
260 '*** called by STOP ***
270 SET PAGE 0,0:COLOR 15,4,7
```

4.6.9 YJK方式とVDPのレジスター

YJK方式による表示を、BASICのSCREEN 10~12の代わりに、マシン語プログラムがVDPのレジスターを操作して行なう方法を紹介しよう。コントロールレ

ジスター 25 のビット 3 “YJK” とビット 4 “YAE” で、RGB 方式による画面表示とYJK 方式による表示を選択できる。

まずレジスター 25 以外のレジスターを、SCREEN 8 の場合と同様に設定しよう。BASIC 言語では、スクリーンモードの 10 から 12 を指定することで、YJK 方式を選択するわけだ。でも VDP の機能としては、SCREEN 8 の画面が RGB 方式と YJK 方式に切り替えられる、と考えたほうがわかりやすい。ビット 3 の YJK が 0 で、ビット 4 の YAE も 0 なら、SCREEN 8 そのものが設定されるわけだ。そして、ほかのレジスターはそのままで、ビット 3 の YJK を 1 に切り替えると、SCREEN 12 と同じ YJK 方式の表示が設定される。

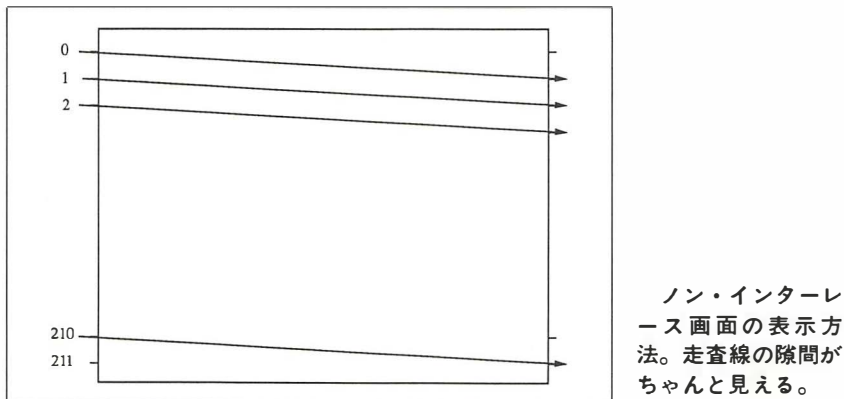
SCREEN 10 や 11 の、YJK と RGB の混在方式での画面を表示するためには、YJK に 1 、YAE にも 1 を設定すればいい。なお、YJK が 0 の場合には、スプライトの色がパレットの影響を受けることはないけど、YJK が 1 だとパレットで変更することができる。

4.7 走査線割り込みを研究する

4.7.1 モニター画面を表示する仕組みは？

MSX2 の SCREEN5 の画面が、横 256 ×縦 212 ドットの点 (ピクセル) の集まりで表わされることは知ってるよね。これを実際にモニターに表示する仕組みが、図 4.8 に示したもの。画面の左上から右下に向け、1 ラインずつ順番に各ピクセルのデータを送ることで、画面を表示するというわけだ。このとき、横に並んだ 256 個のピクセルの集まりが、**"走査線"** と呼ばれるもの。つまり、横に 256 個のピクセルが集まって走査線ができ、縦に 212 本の走査線が集まって画面ができる、という仕組みになっている。

図 4.8: テレビ画面上の走査線のようす

ノン・インターレース画面の表示方法。走査線の隙間がちゃんと見える。

コンピューターの画面だけでなく、普通のテレビ放送なども、この走査線を使って表わされている。日本やアメリカの **"NTSC"** 方式というテレビ放送では、525 本の走査線で画面が表示されているぞ。ただし実際に画面に映る数は約 490 本。残りの走査線には、**"同期信号"** と呼ばれるテレビを制御するための信号や、文字多重放送のための信号が含まれている。1 秒間に表示される画面の数は 30 枚、1 画面には 525 本の走査線が含まれるわけだから、525 × 30 = 15750 本もの走査線が、たった 1 秒間の画面に表示されているわけだ。テレビ放送や MSX の画面表示の仕様にある、"垂直走査周波数 30Hz 、水平走査周波数 15.75kHz" という値は、こうした意味を持っている。

16 ビットコンピューターなどに多く見られる、横 640 ×縦 400 ドットの画面表示機能を持つものでは、多くのドットを表示するために水平走査周波数が 24kHz の、専用モニターが必要になる。また最新のコンピューターではさらに画面表示が細か

くなり、31.5kHzや34kHzのモニターが使われる。

こうした、さまざまな水平走査周波数の画面にも対応したものが、**“マルチ・スキャン・モニター”**。15.75kHzから34kHzまでのどの周波数にも対応したものから、15.75kHzと24kHzの2種類を切り替えるものなど、さまざまな機種がある。これから新しくモニターを買うなんて場合は、自分が持っているコンピューターや将来使いたいコンピューターの仕様をよく調べて、多くの水平走査周波数に対応するマルチ・スキャン・モニターを選ぼう。

参考までに書いておくと、西ヨーロッパ諸国では**“PAL”**、フランスやソ連などでは**“SECAM”**という方式のモニターが使われ、これらの方式ではNTSC方式と走査線の数が異なる。だから、ヨーロッパ製のコンピューターや輸出用のMSXを日本のモニターに接続しても、画面を表示することはできない。ソニーやビクターなどではNTSC、PAL、SECAMの各方式を切り替えられるモニターを作っているけど、輸出用コンピューターの検査などといった特殊な用途に使われるものなので、非常に高価だ(最近、パナソニックから、世界各国のビデオを再生できるビデオデッキが発売されたようだ)。

また、これも余談になるけど、VTRやビデオディスクの性能を示す**“水平解像度”**というものは、走査線の数とは関係ない。コンピューター画面の横方向のドット数に相当する、画面の細かさを表わしたもの。垂直解像度は走査線の数と同じで、どのVTRでも同じ。けれども、**“EDβ”**や“SVHS”では、従来のVTRよりも水平解像度が良くなっている。

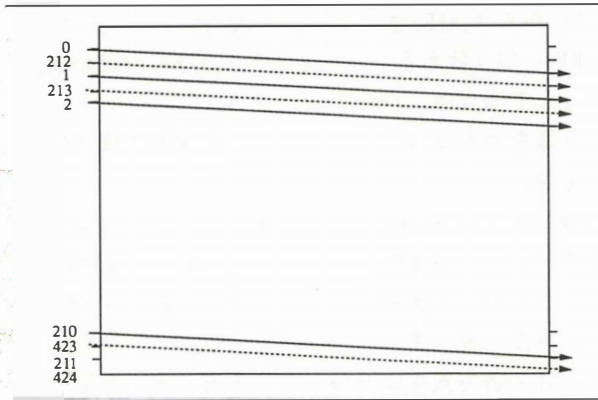
4.7.2　インターレース方式によるテレビ放送

コンピューター画面を表示する仕組みは図4.8のようだったけど、これがテレビ放送ということになると、厳密にはちょっと違ってくる。まずは図4.9をじっくりと見てほしい。

これは**“インターレース”**という画面の表示方式を説明したもの。実際のテレビ放送では、この方式が採用されているわけだ。まず画面上端の走査線を0番とすると、1番、2番、3番と、図4.8で説明したのと同じように走査線を表示していく。そして画面の一番下までいったら、今表示した各走査線の間を埋めるように212番、213番、214番と、順番に走査線を表示していくわけだ。

インターレースという単語を直訳すると、“織り交ぜる”という意味。図4.8で説明した走査線の間を、さらにもう1本の走査線で埋めることで、ピクセル間の隙間をなくしているんだ。ちなみに、図4.8のように、インターレースによらずに全部の走査線を順番に表示する方式を、**“ノン・インターレース”**という。

それでは、テレビ放送でなぜこのインターレース方式が採用されたかといえば、

図 4.9: インターレースモードではこうなるぞ

こちらはインターレース画面の表示方法。走査線の隙間が見えなくなる。

さまざまな原因から発生する雑音(ノイズ)と、それにより起きる画面の乱れを目立たなくするため。たとえば、0番と1番の走査線が雑音によって乱れても、多少の時間をおいてから、その間に表示される212番の走査線が正常ならば、画面の乱れも目立たないというわけだ。電波が空中を飛んでくる間に受ける雑音や、ほかの電化製品の影響などで発生する雑音、電源コンセントから拾う雑音など、テレビ放送がさまざまな雑音から影響を受けやすいだけに、このインターレースというのは有効な方式なんだ。

これとは逆にインターレース方式の欠点は、となりあう走査線の位置がわずかにずれているために、画面がゆれて見えること。テレビ放送のように動きがある画面ではさほど気にはならないけど、コンピューター画面のように細かい文字を静止した状態で表示する場合などには、意外とちらつきが目立ちやすい。このため、MSXも含めて、多くのコンピューターでは、ノン・インターレースでの画面表示が通常は使われている。

4.7.3 MSX2におけるインターレース画面

多くのコンピューターでは、“通常は” ノン・インターレース画面が使われる…━と条件付きで書いた理由は、MSX2以降ではインターレース画面も使えるから。

どうして、ちらつきの多いインターレース方式を採用したかという第1の目的は、家庭用のテレビモニター(つまりコンピューター専用でない、いわゆる家庭用テレビと呼ばれるもの)で、縦424ドットの画面表示を行なうためだ。MSX2が開発された1985年当時は、マルチ・スキャン・モニターが一般的でなく、また水平走査周波数24kHzのモニターも10万円を越える高価なものだった。そこで、MSX2と組み

合わせるモニターとしては、一般の家庭用テレビや、水平走査周波数 15.75kHz の低解像度モニターが主流となっていたわけだ。たとえば MSX2+で、

```
CALL KANJI2
```

という命令を実行し、画面をインターレースモードに切り替えることで、縦 24 行の漢字表示を可能にしたわけだ。ただし、前にも書いたように、インターレースでの画面表示はちらつきが多く目が疲れやすいので、時々休憩するように心がけよう。

第 2 の目的は、第 1 の目的よりも積極的なもので、MSX2 とテレビ画面を接続することによる **“スーパーインポーズ”** や **“ビデオ・デジタイズ”** を可能にするためだ。水平走査周波数 15.75kHz と、インターレース方式を使う MSX2 の画面は、テレビ画面にコンピューターの画面を重ねるスーパーインポーズや、テレビ放送やビデオカメラの画像をコンピューターに取り込む、ビデオ・デジタイズに最適。コンピューター本体にこうした機能をはじめから内蔵したパナソニックの FS–5500 や、オプションのボードを接続することで可能となるビクターの HC–95 などは、ずいぶん前に発売されたマシンながら、いまでも画像データの取り込みや加工といった目的に活躍しているという。

そしてインターレース方式の第 3 の利点は、写真写りがよいこと。ノン・インターレースの画面写真では走査線の隙間が見え、印刷すると **“モアレ”** という縞模様が現われやすい。でもインターレース画面では走査線の隙間がなく、モアレが出る心配もないわけだ。

MSX2 以降のマシンでは、つぎのように、画面をインターレースモードに切り替えることができる。ただし漢字モードを指定できるのは、MSX2+以降だけだ。

MSX2 マシンの場合	`SCREEN ,,,,,1`
MSX2+マシン以降の場合	`CALL KANJI3`

4.7.4 走査線割り込みの原理を探る

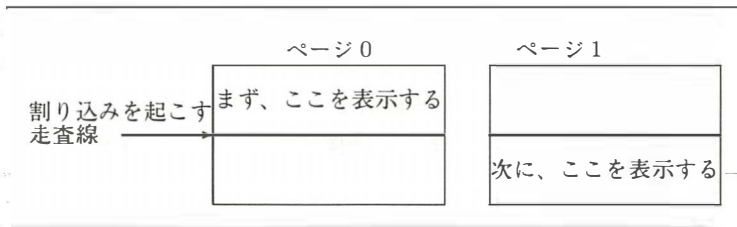
“割り込み” とは、例外的な条件によってプログラムの流れを変えること。たとえばジョイスティックのボタンが押されたら、BASIC のプログラムの流れを変えるための、“`ON STRIG GOSUB`” という命令も、この割り込みを処理する命令の一種なんだ。

“走査線割り込み” もこれと原理は同じで、指定された番号の走査線の表示が終わると、割り込み処理が行なわれるというもの。でも、走査線割り込みは高速に処理される必要があるので、BASIC でプログラムしていたのでは間に合わない。マシン語プログラムによる割り込み処理が必要になる。

それでは、割り込み処理の原理を簡単に説明する。まず画面表示を制御するVDPは、特定の条件によって、CPUに割り込み信号を送ることができる。この条件には、“ライトペン”、“垂直帰線”、“走査線”という3種類があるけど、MSX2ではライトペンの割り込みは使われない。

まず**“垂直帰線割り込み”**とは、画面の下端の表示が終わり、次の画面の表示を準備しているときに発生する割り込みで、1/60秒ごとかならず発生する。これが、ゲーム中の音楽の演奏タイミングの調整などに使われる、MSXの**“タイマー割り込み”**の正体だ。そして今回問題となるのが、特定の走査線の表示が終わったときに発生する、走査線割り込みだ。この割り込みも、1/60秒ごとに発生する。

図 4.10: 走査線割り込みの原理なのだ

ページ0
ページ1
割り込みを起こす走査線
まず、ここを表示する
次に、ここを表示する

垂直帰線割り込みと走査線割り込みを組み合わせると、画面の上端と任意の走査線の2ヵ所で、割り込みを発生させることができる。これで、画面を垂直帰線割り込みから走査線割り込みまでの上部分と、走査線割り込みから垂直帰線割り込みまでの下部分に、2分割できるというわけだ。

またMSX2では、“ページ”と呼ばれる複数の画面を持つことができる。ビデオRAMが128キロバイトのMSX2では、SCREEN 5または6で4画面、SCREEN 7と8では2画面のページを持てるというわけ。これらをBASICの、“`SET PAGE`”命令で切り替えれば、複数の画面を瞬時に表示することができる。

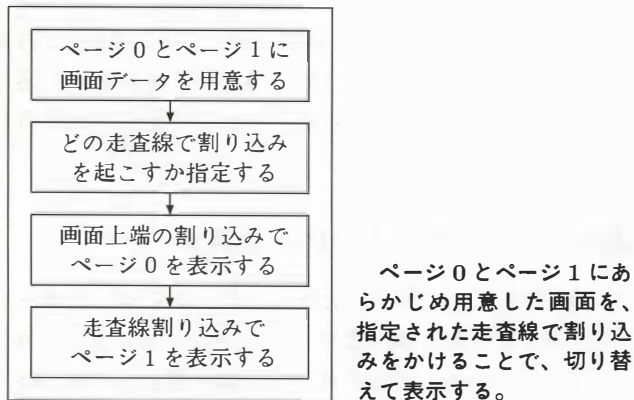
この機能を利用したのが、ゲームソフトなどで活用されている走査線割り込み。マシン語で割り込み処理プログラムを作り、走査線割り込みでページを切り替えることで、画面の上側と下側にべつべつのページを表示する。さらに、VDPのスクロール機能を組み合わせれば、片方の画面だけをスクロールさせることも可能だ。

4.7.5 走査線割り込みの実例を紹介する

筆者は正直にいって、はじめてVDPの仕様書を見たとき、“走査線割り込みなんか何の役に立つのだろう”と疑問を持ったものだった。“MSX2テクニカルハンドブック”などにも、走査線割り込み機能があるということは紹介されていたものの、

図 4.11: 走査線割り込みの手順

ページ 0 とページ 1 に画面データを用意する
↓
どの走査線で割り込みを起こすか指定する
↓
画面上端の割り込みでページ 0 を表示する
↓
走査線割り込みでページ 1 を表示する

ページ 0 とページ 1 にあらかじめ用意した画面を、指定された走査線で割り込みをかけることで、切り替えて表示する。

具体的なプログラム例や応用例は掲載されていなかった。それが実際にゲームソフトに応用され、だれもをウーンとうならせたのは、コンパイルが開発しポニーキャニオンから発売された、"ザナック EX" というゲームが登場してからだと思う。

MSX が MSX2 になって拡張された機能のひとつに、"ハードウェア縦スクロール" がある。これはシューティングゲームを作るには非常に便利な機能だったのだけど、そのままでは画面全体がスクロールしてしまうため、ゲームには欠かせないスコア表示などを、画面上に固定することができなかった。ところが "ザナック EX" では、画面を高速に縦スクロールさせながら、画面上端の固定位置にスコアを表示させていたのだ。当時の M マガ編集部では、どういうワザを使っているのか話題になったけど、スコア部分とスクロール部分の境界のちらつきから、走査線割り込みと判明した。と、まあ、一度気付いてしまえば "コロンブスの卵" で、これ以降、走査線割り込みを利用したシューティングゲームが、次々と開発されるようになる。走査線割り込みを使ったゲームソフトを、ポーズキーで停止させると、ふたつの画面のうちどちらかしか表示されない。実際に試してみよう。

さらに MSX2+では、走査線割り込みと "ハードウェア横スクロール" を組み合わせて、画面の一部分だけを横スクロールさせることも行なわれている。コナミから発売された "F–1 スピリット 3D スペシャル" では、ゲーム画面だけを横スクロールさせながら、走査線割り込みによって、画面下部に F1 マシンのコックピットのようすを表示しているようだ。

4.7.6　いよいよ実践編はりきっていこう！

実際に走査線割り込みを使ったプログラム例を紹介しよう。詳しくは後で説明するけど、このプログラムはマシン語で作られたもの。でも、BASIC言語から呼び出して使えるようになっているので、自作のゲームなどにもドンドン応用できるはずだ。また、ソースリストも公開しておくので、アセンブラーがわかる人は頑張って解析してみてほしい。

4.7.7　走査線割り込みに使うVDPレジスター

走査線割り込みを起こすための手順は次のとおり。まずコントロールレジスター19に割り込みを発生させたい走査線の番号を書き込み、コントロールレジスター0のビット4を1に変える。すると、指定された走査線の表示が終わったときに、VDPがCPUに対して割り込みをかける。また、割り込みがかかったときに、ステータスレジスター1のビット0が1であれば、割り込みの原因が走査線割り込みであることがわかる。これらのことをまとめたのが、図4.12と4.13だ。

図4.12: 走査線割り込みを発生するVDPレジスター

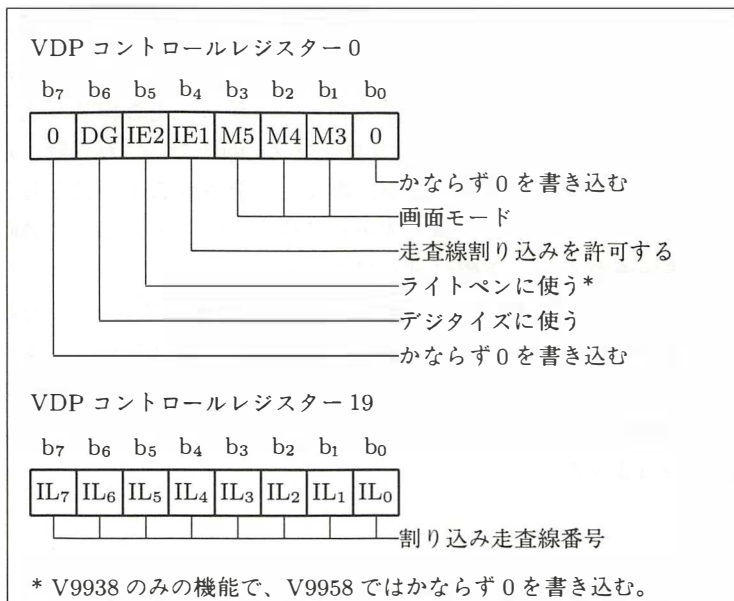
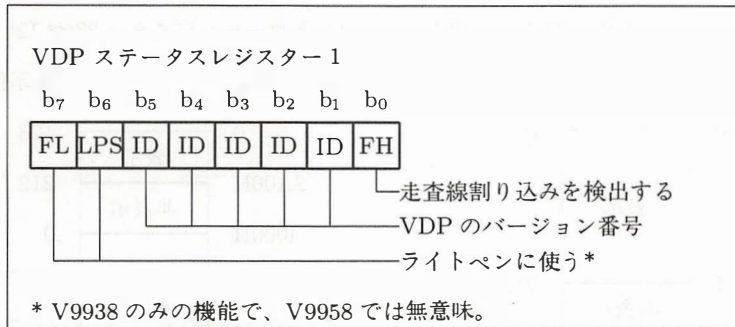



図 4.13: 走査線割り込みを検出する VDP レジスター

とまあ、以上が走査線割り込みに直接関係するレジスターの説明。それでは、べつのレジスターを使って、画面の切り替えとハードウェア縦スクロールを制御してみよう。

SCREEN 5 または SCREEN 6 で、コントロールレジスター 2 のビット 6 と 5 にページ番号を書き込むと、BASIC の “SET PAGE” 命令と同様に、ディスプレーページ (画面に表示するページ) を切り替えることができる (図 4.14 参照)。ビット 7 には 0 を、ビット 4 から 0 には 1 をそれぞれ書き込むわけだ。これと同じように SCREEN 7 や SCREEN 8 の場合は、ビット 5 でページを指定し、ビット 6 にはかならず 0 を

図 4.14: 画面切り替えを制御する VDP レジスター

VDP コントロールレジスター 2

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	A_{16}	A_{15}	1	1	1	1	1

b_4–b_0: かならず 1 を書き込む
b_6–b_5: ディスプレーページ
b_7: かならず 0 を書き込む

VDP コントロールレジスター 23

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
DO_7	DO_6	DO_5	DO_4	DO_3	DO_2	DO_1	DO_0

b_7–b_0: ディスプレーオフセット

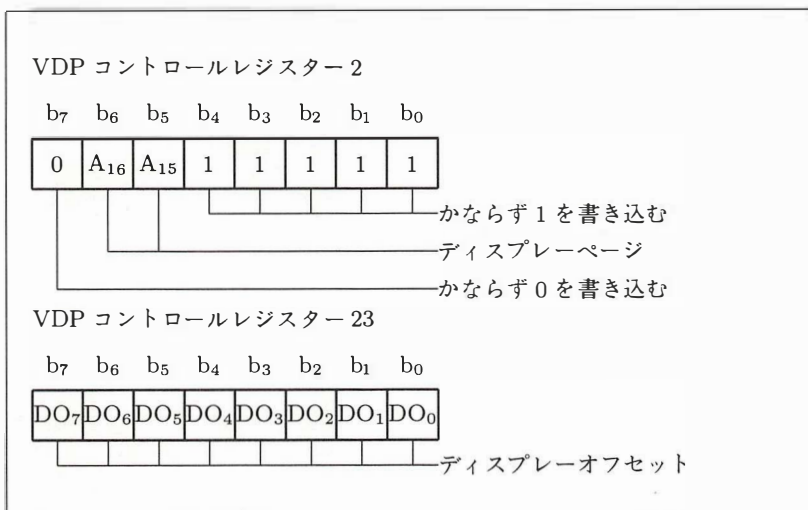



図 4.15: ハードウェア縦スクロールの仕組み

コントロールレジスター 23 が 0 の場合

VRAM 番地		表示位置
0	表示	0
6A00H	非表示	212
7FFFH		255

コントロールレジスター 23 が 128 の場合

VRAM 番地		表示位置
0	表示	128
2A00H	非表示	212
4000H	表示	0
7FFFH		128

書き込むことで、ページを切り替えることが可能になる。

また、コントロールレジスター 23 を使うと、ハードウェア縦スクロールを行なうことができる。具体的には図 4.15 のように、レジスターに設定した値によって画面の表示位置とビデオ RAM の番地の対応が変わり、画面が縦スクロールするという仕組みだ。ただし、このハードウェア縦スクロールを使うと、割り込みを起こす走査線の番号もずれてしまうので、走査線番号に縦スクロール量を加えるなどの補正が必要になる。サンプルプログラムでは、“ON_VSYNC” からはじまるサブルーチンで、この補正をしている。

4.7.8　アセンブルの方法と BASIC 部分の動作

走査線割り込みを起こすためのサンプルプログラムは、アセンブラーで作られた部分と BASIC 言語で作られた部分からできている。アセンブラーの部分の動作原理は次に説明するとして、まずはアセンブルの方法と、BASIC 部分の動作原理から紹介しよう。

アセンブラーのプログラム (ソースリスト) 自体は、後のページのリスト 4.9 に掲載しておいた。これを MSX–DOS のスクリーンエディターなどで入力し、“ONSCAN.Z80” というファイル名でセーブしよう。次に “MSX–DOS TOOLS” に入っている、“M80.COM” と “L80.COM” というプログラムを使い、次の手順どおりにアセンブルとリンクをする。“ONSCAN.BIN” というマシン語ファイルができれば完了だ。なお “DEL ONSCAN.BIN” という部分は、アセンブルをやり直すときに古いファイルを消すためのもの。一度目は不要のものだ。

```
M80 ,=ONSCAN.Z80/R/Z
L80 ONSCAN,ONSCAN/N/E
DEL ONSCAN.BIN
REN ONSCAN.COM ONSCAN.BIN
```

それでは次に、マシン語ファイルをコントロールする、BASIC プログラム (リスト 4.8 参照) の動作原理を説明する。

まず 190 行から 200 行の部分は、ビデオ RAM のページ 0 を初期化するためのもの。画面を縦スクロールさせる場合にはページ 0 全体を初期化する必要があるのだけど、“CLS” コマンドでは画面に表示される部分だけしか初期化することができない。そこで “COPY” 命令を使い、画面の (0,0)–(255,127) の内容を (0,128)–(255,255) に複写し、ページ 0 全体を初期化しているというわけだ。また 210 行、240 行、250 行は、それぞれテストパターンを表示する部分。

BASIC のプログラムからマシン語のプログラムを呼び出すには、**“USR 関数”** を利用する。260 行は割り込みを起こす走査線を指定し、画面上側にページ 0 を、画面下側にページ 1 を表示させる部分だ。

```
USR(256+走査線番号)
```

という書式にしたがって、走査線番号の値を変えると、割り込みを起こす場所が変化する。

さらに、画面上側に表示されているページ 0 の部分を、縦スクロールさせているのが 290 行。

```
USR(512+走査線番号)
```

という書式で値を指定する。また、今回は使っていないけど、

```
USR(768 ＋走査線番号)
```

を指定すると、画面下側に表示されるページ 1 の部分を縦スクロールさせることができる。走査線割り込みを中止するには、

```
USR(0)
```

を実行すればいい。

なおこのプログラムでは、アセンブラー部分を簡単にするため USR 関数のパラメーターを整数に限っている。だから、

```
USR(868.0)
```

とか

```
USR(100+A!)
```

のような、実数のパラメーターは使えない。

リスト 4.8 (ONSCAN.BAS)

```
100 '
110 ' onscan1.bas : test interrupt on scan
120 ' by nao-i on 24. Sep. 1989
130 '
140 CLEAR 300,&HAFFF
150 DEFINT A-Z
160 OPEN "GRP:" FOR OUTPUT AS #1
170 BLOAD "onscan.bin"
180 SCREEN 5
190 SET PAGE 0,0: COLOR 15, 6,1: CLS
200 COPY (0,0)-(255,127) TO (0,128)
210 CIRCLE (128,106),80: PAINT (128,106)
220 SET PAGE 1,1: COLOR 15, 1,1: CLS
230 DEFUSR0=&HB000
240 PSET (8,192),0,PRESET
250 PRINT #1,"This area will not scroll."
260 JK=USR(256+100)
270 FOR J=1 TO 5
280   FOR I=0 TO 255
290     JK=USR(512+I)
300   NEXT I
310 NEXT J
320 JK=USR(0): COLOR 15,4,4: SCREEN 0
330 END
```

4.7.9 アセンブラー部分の動作原理だ

それではいよいよ、リスト 4.9 の、実際に走査線割り込みを処理するメインプログラムの動作原理を解説する。

まず、“ASEG” からの 3 行は、M80.COM や L80.COM を使って BASIC のサブルーチンなどの特殊なオブジェクトを作るための命令。M80.COM と L80.COM は通常ペアにして使い、アセンブラーで作られたプログラムからマシン語のファイルを作り出すものだ。ただ、このとき作られるファイルの拡張子は “COM” 。つまり MSX–DOS 上で実行可能なマシン語ファイルになってしまう。そこで今回のように BASIC で扱えるファイルを作りたい場合には、この ASEG からの命令が必要になるというわけだ。

また、BASIC で BLOAD できる形式のマシン語プログラムの先頭には 7 バイトのヘッダーがあり、

FEH
ロード開始番地
ロード終了番地
実行開始番地

といったデータがそれぞれ書き込まれている。これらを指定しているのが、ASEG の次の 4 行だ。

さて、“AD_LOAD:” からうしろの部分が、プログラムの本体になる。最初に行なっているのが、USR 関数のパラメーターの処理。パラメーターが整数ならば A レジスターの内容が 2 になるので、それを確認している。また、入力されたパラメーターの値は、HL+2 番地と HL+3 番地に記録される。このとき、パラメーターの上位バイトが 0 ならば後始末、1 ならば走査線割り込みの設定、2 ならばページ 0 の縦スクロール、3 ならばページ 1 の縦スクロールを行なうというわけ。

実際に割り込みが発生すると、FD9AH 番地 (HOOKDT の部分) がコールされることになる。ここからの 5 バイトに、

```
RST   30H
DB    スロット番号
DW    番地
RET
```

を書いておくと、割り込みが発生したときに、指定したプログラムを呼び出すことができる。このように、何かの条件でコールされる場所を **“フック”** という。ここでは、“ON_H.KEYI:” が呼び出されるように準備している。

FD9FH 番地からはタイマー割り込み (ON_H.TIMI:) 、つまり垂直帰線割り込みを呼び出す部分。ここでは、割り込みが発生したときの VDP ステータスレジスター

0の値がAレジスターに記憶されるので、AFレジスターを書き替えてはいけない。もし、どうしてもフックを書き替えるならば、サンプルプログラムのようにフックの元の値を保存しておき、割り込み処理が終わったときに保存したフックへジャンプするようにしよう。

割り込みの発生で呼び出され、VDPを操作するプログラムを，“ON_VSYNC:”と、“ON_SCAN:”からはじまるサブルーチンにまとめておいた。ここを書き替えれば、走査線割り込みをべつの目的に使えるだろう。

サブルーチン“_VDPSTA”は、VDPのステータスレジスターを読む。サブルーチン“WRTVDP”は、VDPのコントロールレジスターに指定された値を書き込んで、その値をそれぞれの保存場所(表4.9参照)に保存する。もっとも、前にも書いたように、MSX2や2+のROMにはこれらのサブルーチンと同じ機能のBIOSがあるので、普通は自分でサブルーチンを作らずにBIOSを使えばいい。でも、これらのBIOSはサブROMにあったり、処理中にサブROMを呼び出したりするので、多少時間がかかるという難点がある。そのため、今回のような割り込み処理には都合が悪いので、あえてBIOSを使わなかった。

これはBASICのマシン語サブルーチンには関係ないことだけど、DOSのプログラムでは割り込み処理プログラムを4000Hよりも大きい番地に置く必要がある。これ以下では特定のスロット構成のMSXで不都合が起きるからだ。

リスト 4.9 (ONSCAN.Z80)

```
;
;       onscan.z80 : test program for interrrupt on scan
;       by nao-i on 26. Sep. 1989
;
;       called as USR function from BASIC
;       USR(&H00xx)     restore registers and hooks
;       USR(&H01xx)     set interrupt line
;       USR(&H02xx)     set display offset line of page 0
;       USR(&H03xx)     set display offset line of page 1
;
        .Z80
START   EQU     0B000H  ; address to load and execute
USE_SUB         EQU     0
USE_WRTVDP      EQU     0
;
        IF      USE_WRTVDP
WRTVDP  EQU     0047H
        ENDIF
EXTROM  EQU     015FH
VDPSTA  EQU     0131H
SETPAG  EQU     013DH
;
RAMAD2  EQU     0F343H          ; slot of RAM in page 2
RAMAD3  EQU     0F344H          ; slot of RAM in page 3
RG0SAV  EQU     0F3DFH
DPPAGE  EQU     0FAF5H
ACPAGE  EQU     0FAF6H
H.KEYI  EQU     0FD9AH
H.TIMI  EQU     0FD9FH
RG8SAV  EQU     0FFE7H
;
        ASEG
        ORG     100H            ; to make .COM file
        .PHASE  START-7
;
        DB      0FEH            ; header to BLOAD
        DW      AD_LOAD         ; address to load
        DW      AD_NEXT-1       ; address of end of file
        DW      DO_NOTHING      ; address to execute
;
AD_LOAD:
        PUSH    AF
        PUSH    HL
        PUSH    DE
        PUSH    BC
        CP      2
        JR      NZ,RESET_SCAN   ; parameter is not integer
        INC     HL
        INC     HL
        LD      E,(HL)
        INC     HL
        LD      D,(HL)          ; DE = parameter of USR()
```

```
        LD      A,D
        OR      A
        JR      Z,RESET_SCAN
        DEC     A
        JR      Z,SET_SCAN
        DEC     A
        JR      Z,SET_D0
        DEC     A
        JR      Z,SET_D1
        JR      RESET_SCAN
;
SET_SCAN:
        LD      A,E
        LD      (ILSAV),A
        CALL    SET_ILREG       ; set interrupt line
        LD      A,(HOOKED)
        OR      A
        JR      NZ,RET_BASIC    ; hook is already set
;
        LD      HL,H.KEYI
        LD      DE,HOOKSA
        LD      BC,10
        LDIR                    ; save hooks
        LD      HL,HOOKDT
        LD      DE,H.KEYI
        LD      BC,10
        DI
        LDIR                    ; set hooks
        LD      A,(RAMAD2)
        LD      (H.KEYI+1),A
        LD      (H.TIMI+1),A
        LD      A,(RG0SAV)
        OR      00010000B
        LD      B,A
        LD      C,0
        CALL    WRTVDP          ; interrupt on
        LD      A,1
        LD      (HOOKED),A
        JR      RET_BASIC
;
RESET_SCAN:
        CALL    RESET_SCAN_SUB
        JR      RET_BASIC
;
SET_D0:                         ; set display offset of page 0
        LD      A,E
        LD      (D0VAL),A
        JR      RET_BASIC
SET_D1:                         ; set display offset of page 1
        LD      A,E
        LD      (D1VAL),A
;
RET_BASIC:
        EI
        POP     BC
```

```
        POP     DE
        POP     HL
        POP     AF
DO_NOTHING:
        RET
;
RESET_SCAN_SUB:
        DI
        LD      A,(RG0SAV)
        AND     11101111B
        LD      B,A
        LD      C,0
        CALL    WRTVDP          ; inturrupt off
        LD      BC,23           ; write 0 into reg#23
        CALL    WRTVDP          ; restore display offset
        XOR     A
        LD      (DPPAGE),A
        LD      BC,1F02H        ; write 1FH into reg#2
        CALL    WRTVDP          ; set page 0
        LD      A,(HOOKED)
        OR      A
        RET     Z
        LD      HL,HOOKSA
        LD      DE,H.KEYI
        LD      BC,10
        LDIR                    ; restore hooks
        XOR     A
        LD      (HOOKED),A
        RET
;
SET_ILREG:
        LD      A,(ILSAV)
        LD      HL,DOVAL
        ADD     A,(HL)          ; interrupt line = (ILSAV) + (DOVAL)
        LD      B,A
        LD      C,19
        JP      WRTVDP          ; write interrupt line # into reg#19
;
ON_H.KEYI:                      ; called from H.KEYI
        LD      A,1
        CALL    _VDPSTA         ; read status reg#1
        AND     1
        CALL    NZ,ON_SCAN
        JR      HOOKSA
;
ON_H.TIMI:                      ; called from H.TIMI
        PUSH    AF
        CALL    ON_VSYNC
        POP     AF              ; do not change AF in H.TIMI
        EI
        JR      HOOKSA+5
;
HOOKDT:
        RST     30H
        DB      0
```

```
        DW      ON_H.KEYI
        RET
        RST     30H
        DB      0
        DW      ON_H.TIMI
        RET
;
;       data area
;
HOOKED: DB      0       ; non-zero if hooks have been set
HOOKSA: DS      10      ; save area for hooks
DOVAL:  DB      0       ; display offset of page 0
D1VAL:  DB      0       ; display offset of page 1
ILSAV:  DB      0       ; interrrupt line
;
;       _VDPSTA : read a VDP status register
;       Entry   A       VDP status register #
;       Return  A       value of the status register
;       Modify  AF, BC
;       Note    compatible with ROM-BIOS
;               DI when return
;
_VDPSTA:
        IF      USE_SUB
        LD      IX,VDPSTA
        JP      EXTROM
        ELSE
        DI
        AND     00001111B
        LD      B,A
        LD      A,15
        LD      C,A
        CALL    WRTVDP
        LD      BC,(6)
        INC     C
        IN      A,(C)
        PUSH    AF
        LD      BC,15
        CALL    WRTVDP
        POP     AF
        RET
        ENDIF
;
;       WRTVDP : write a byte into VDP register
;       Entry   B       datum to write
;               C       VDP register #
;       Return  none
;       Modify  AF, BC
;       Note    compatible with ROM-BIOS
;               DI when return
;
        IFE     USE_WRTVDP
WRTVDP:
        PUSH    HL
        PUSH    DE
```

```
        LD      D,B             ; =datum
        LD      A,C             ; =register #
        LD      HL,RGOSAV
        CP      8
        DI
        JR      NC,SAVEREG
        LD      HL,RG8SAV-8
        CP      24
        JR      NC,NOSAVE
SAVEREG:
        XOR     A
        LD      B,A             ; BC=register #
        ADD     HL,BC           ; HL=RG?SAV
        LD      (HL),D          ; save datum
NOSAVE:
        LD      A,C             ; =register #
        LD      BC,(7)
        INC     C
        OUT     (C),D
        AND     00111111B
        OR      10000000B
        OUT     (C),A
        POP     DE
        POP     HL
        RET
        ENDIF                   ; IFE USE_WRTVDP
;
;       please modify following subroutines as you need
;
ON VSYNC:
        XOR     A
        LD      (DPPAGE),A
        LD      BC,1F02H        ; write 1FH into reg#2
        CALL    WRTVDP          ; set page 0
        LD      A,(DOVAL)
        LD      B,A
        LD      C,23
        CALL    WRTVDP          ; set display offset
        JP      SET_ILREG       ; set interrupt line
;
ON_SCAN:
        LD      A,1
        LD      (DPPAGE),A
        LD      BC,3F02H        ; write 3FH into reg#2
        CALL    WRTVDP          ; set page 1
        LD      A,(D1VAL)
        LD      B,A
        LD      C,23
        JP      WRTVDP          ; set display offset
;
AD_NEXT EQU     $               ; end of program + 1
        .DEPHASE
;
        END
```

4.7.10 走査線割り込みのマシン語ルーチンだ

最後に、走査線割り込みのためのプログラムのソースリストを打ち込むのが面倒な人や、アセンブラーを持っていない人のために、自動的にマシン語ファイルを作る BASIC プログラムを掲載する。以下のプログラムを打ち込み、実行させると、自動的に “ONSCAN.BIN” というファイルを作成してくれる。

リスト 4.10 (MKONSCAN.BAS)

```
10 CLEAR 100,&HCFFF:DIMD(15)
20 PRINT"Making onscan.bin":AD=&HB000:C=0:L=0
30 FOR I=0TO15:READ A$:IF A$="*" GOTO100
40 A=VAL("&h"+A$):C=(C+A) AND 255:D(I)=(D(I)+A) AND 255
45 POKE AD,A:AD=AD+1:NEXT
50 READA$:A=VAL("&h"+A$):L=L+1
55 IF C<>A THEN PRINT "Error in line ";990+10*L:END
60 GOTO 30
100 '
110 PRINT"Saving"
120 BSAVE"onscan.bin",&HB000,&HB14D
130 PRINT"Done.":END
1000 DATA F5,E5,D5,C5,FE,02,20,53,23,23,5E,23,56,7A,B7,28, 5D
1010 DATA 4A,3D,28,08,3D,28,49,3D,28,4C,18,3F,7B,32,D9,B0, 00
1020 DATA CD,A1,B0,3A,CC,B0,B7,20,41,21,9A,FD,11,CD,B0,01, 33
1030 DATA 0A,00,ED,B0,21,C2,B0,11,9A,FD,01,0A,00,F3,ED,B0, B0
1040 DATA 3A,43,F3,32,9B,FD,32,A0,FD,3A,DF,F3,F6,10,47,0E, 20
1050 DATA 00,CD,F4,B0,3E,01,32,CC,B0,18,0F,CD,70,B0,18,0A, B4
1060 DATA 7B,32,D7,B0,18,04,7B,32,D8,B0,FB,C1,D1,E1,F1,C9, 61
1070 DATA F3,3A,DF,F3,E6,EF,47,0E,00,CD,F4,B0,01,17,00,CD, E0
1080 DATA F4,B0,AF,32,F5,FA,01,02,1F,CD,F4,B0,3A,CC,B0,B7, 54
1090 DATA C8,21,CD,B0,11,9A,FD,01,0A,00,ED,B0,AF,32,CC,B0, 67
1100 DATA C9,3A,D9,B0,21,D7,B0,86,47,0E,13,C3,F4,B0,3E,01, 2F
1110 DATA CD,DA,B0,E6,01,C4,32,B1,18,13,F5,CD,1C,B1,F1,FB, BA
1120 DATA 18,10,F7,00,AE,B0,C9,F7,00,BA,B0,C9,00,00,00,00, 2A
1130 DATA 00,00,00,00,00,00,00,00,00,00,F3,E6,0F,47,3E,0F, A6
1140 DATA 4F,CD,F4,B0,ED,4B,06,00,0C,ED,78,F5,01,0F,00,CD, E7
1150 DATA F4,B0,F1,C9,E5,D5,50,79,21,DF,F3,FE,08,F3,30,07, EB
1160 DATA 21,DF,FF,FE,18,30,04,AF,47,09,72,79,ED,4B,07,00, 5D
1170 DATA 0C,ED,51,E6,3F,F6,80,ED,79,D1,E1,C9,AF,32,F5,FA, F3
1180 DATA 01,02,1F,CD,F4,B0,3A,D7,B0,47,0E,17,CD,F4,B0,C3, E7
1190 DATA A1,B0,3E,01,32,F5,FA,01,02,3F,CD,F4,B0,3A,D8,B0, 0D
1200 DATA 47,0E,17,C3,F4,B0,*
```

第5章 MSX-MUSIC

この章は、MSX マガジン 1990 年 7 月号から 1990 年 10 月号までの“MSX2+テクニカル探検隊” の記事を再編集したものである。

5.1 FM 音源ってどんなもの

MSX–MUSIC という名称で仕様が定まった FM 音源。ゲームの効果音を迫力あるものにしてくれるのは知っているけど、どんな仕組みになっているのか？ このページではその謎に迫ってみる。

5.1.1 FM 音源へと至る電子楽器の歴史

FM 音源を解説する前に、電子楽器の歴史を振り返ってみよう。

ボグ・ムーグ博士は、電圧で音階を制御できる発振器と、電圧で音色を制御できるフィルターを組み合わせて、**“ムーグ式シンセサイザー”** という楽器を作った。1968 年にはこれを使用した最初のレコードが発表され、1970 年代になると多くの音楽家がシンセサイザーを使うようになった。ただこのシンセサイザーは、最近主流となっている **“デジタルシンセサイザー”** とは違っていて、トランジスタなどのアナログ回路の組み合わせで作られたもの。デジタルに対して **“アナログシンセサイザー”** とも呼ばれている。

でも、このアナログシンセサイザーには、いくつかの欠点があった。それが、温度変化に弱い、高価である、雑音が入りやすい、ということ。筆者も 1970 年代に秋葉原で IC を買って、シンセサイザーを自作したけど、調整が難しかったことが印象に残っている。

さて、そんな欠点を克服するために開発されたのが、デジタル回路による電子楽器。もっとも単純なデジタル音源は、**“プログラマブル・サウンド・ジェネレーター”**、略して **“PSG”** だ。これは、4 個程度のデジタル発振器の出力を、デジタル・アナログ (D/A) コンバーターで、オーディオ信号に変えて出力する LSI 。価格が安く、使いやすいこともあって、MSX などの多くのパソコンに組み込まれている。

PSG よりも複雑な音を作る方法のひとつに、“サンプリング音源” がある。これは、ほかの楽器の音をマイクで受け取って、A/D(アナログ・デジタル) コンバーターでデジタル信号に変え、メモリーに記憶し、D/A コンバーターでアナログ信号に戻して再生するもの。これを応用した楽器が、**“サンプリングシンセサイザー”** というわけだ。音を作る自由度は高いけど、大量のメモリーを必要とするなど、ハードウェアが高価になることが欠点といえる。

さて、PSG の安さと、サンプリング音源の自在さを合わせ持った音源として注目されるのが、FM 音源だ。FM とは、FM 放送やモデムの FM 信号と同じ、“周波数

表 5.1: 電子楽器の性能を比較する

	アナログシンセ	PSG	サンプリングシンセ	FM 音源
ハードウェア	複雑	LSI	LSI+大容量メモリー	LSI
安定性	温度変化に弱い	安定	安定	安定
音色	多彩	貧弱	万能	多彩
データ量		小さい	莫大	小さい
価格	高い	安い	高い	並

変調” という意味。図 5.1 のように、1 個のデジタル発振器の出力が、もう 1 個のデジタル発振器の周波数を変調して、PSG よりも複雑な音を作り出す。1 個のデジタル発振器を **“オペレーター”** ともいい、図 5.1 のように 2 個の発振器を含む FM 音源を、“2 オペレーター式 FM 音源” という。ちなみに **“MSX–MUSIC”** は、正式名称を **“OPLL YM2413”** といい、9 組の 2 オペレーター式 FM 音源を内蔵する LSI だ。

図 5.1: 4 種類の電子楽器の構造を探る

アナログシンセサイザー
アナログ発信器 → アナログフィルター →

PSG
ノイズ音源 →
矩形波音源 →
矩形波音源 →
矩形波音源 →
D/A コンバーター →

この中で、現在の MSX マシンに関係しているのが、PSG と FM 音源。前者は MSX1 のころからお馴染みの音源で、すべてのマシンに内蔵されている。図を見ればわかるように、3 つの音と 1 つのノイズを使って音を作り出す。そして後者は、MSX–MUSIC と呼ばれるもの。FMPAC にも同じ音源が入っている。

サンプリング音源
マイク → A/D コンバーター → メモリー → D/A コンバーター →

FM 音源
デジタル発振器 → デジタル発振器 → D/A コンバーター →
変調

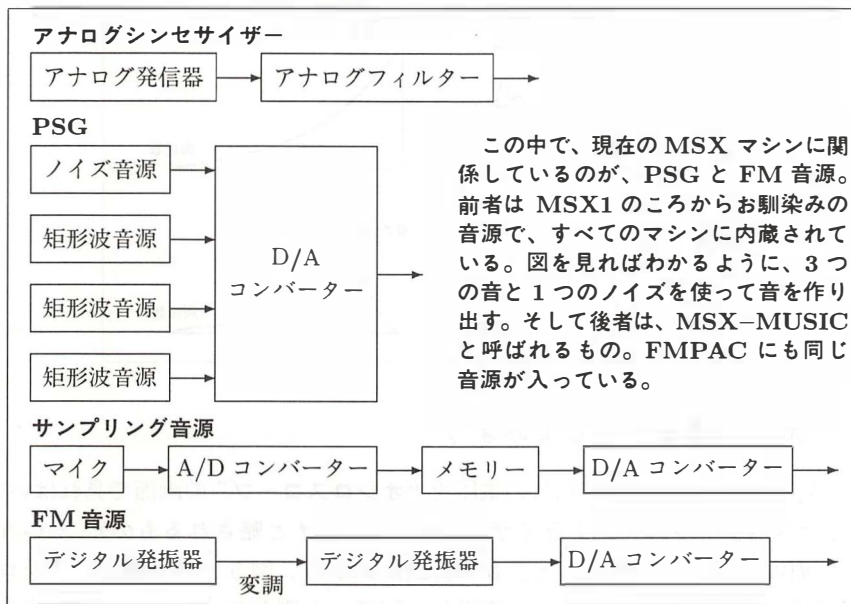
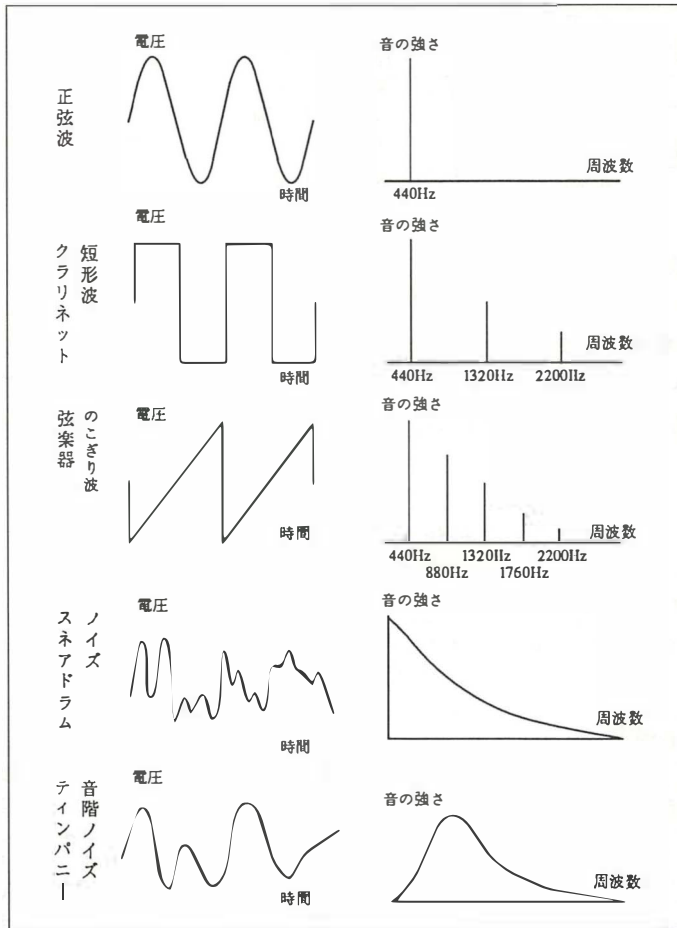



図 5.2: 基本となる音を分析してみる

5.1.2 楽器の音を分析してみよう

音を“見る”ためには、音の信号の電圧を**“オシロスコープ”**の画面で見ればいい。さらに、**“スペクトラム・アナライザー (俗にスペアナと略されるもの)”**という装置で音を周波数成分に分解すると、音の特徴がわかる。図 5.2 の左側は、オシロスコープで見た楽器音の波形の特徴を誇張した図で、右側がスペクトラム・アナライザーで測った周波数成分だ。

もっとも基本的な音は、波形が三角関数の sin で表わされる**“正弦波”**と呼ばれるもの。これはひとつの周波数の音のみを含む。次は波形が四角い**“矩形 (くけい) 波”**で、440Hz 、1320Hz 、2200Hz ……のように、基本周波数とその奇数倍の周波

数の音を含むものだ。実際の楽器では、クラリネットの音がこの矩形波に近い。次に基本的なのは **“のこぎり波”** 。これは、基本周波数とその倍数の周波数の音を含んでいて、弦楽器の音の性質に似たものだ。アナログシンセサイザーは、のこぎり波を加工して、実際の楽器に似た音を作っている。さて、打楽器、とくにスネアドラムの音は、ほかの楽器とは大きく違っている。規則性がなく、どちらかといえば **“ノイズ (雑音)”** に近いものだ。スペクトラム・アナライザーで見ると、広い範囲の周波数の音を含んでいる。

ティンパニーの音は、弦楽器と打楽器の中間の性質で、基本周波数とその近くの周波数の音を含んだもの。**“音階ノイズ”** とも呼ばれている。また、この音階ノイズを加工することで、風、波、口笛などの音も合成できる。体育の授業などで先生が吹く笛や、素人が吹く管楽器の音も、音階ノイズだ。

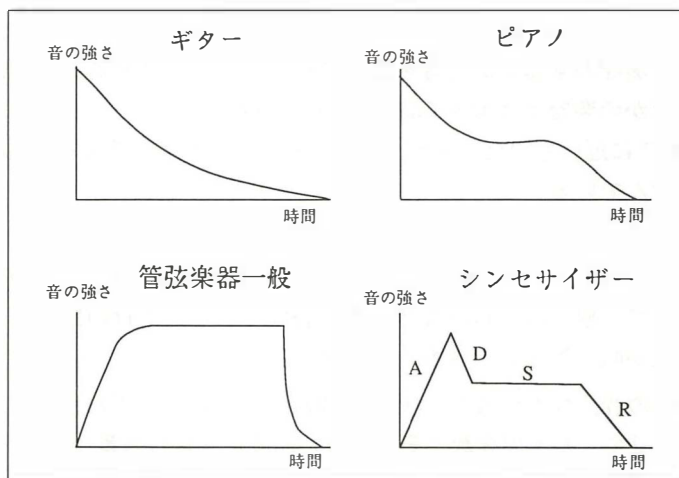
これらの楽器の音、つまり図 5.2 のような波形の代わりに、FM 音源は変調によって正弦波を歪ませて、基本周波数とその倍数の周波数を含む複雑な波形を作り出す。これは難解な技術で、試行錯誤で数値を調整して楽器の音を真似るしかない。そこで、FM 音源にはいくつかの楽器音を合成するためのプログラムが内蔵され、これらの音から選んで使うことが、一般的になっている。

さて、音を特徴づける要素としては、基本となる波形のほかに、音の強弱の変化も挙げられる。この “強弱” は、基本の波形を “包む” という意味で、**“エンベロープ”** と呼ばれるものだ。

たとえば、ギターや打楽器を鳴らすと、その瞬間に強い音が出て、あとは少しずつ音が弱まっていく。ピアノのキーを押すと、はじめは大きな音が出て、それが少しずつ弱まり、キーを押している間はそのままほぼ一定の大きさの音が続く。そしてキーをはなすと、音が弱くなっていく。また一般の管弦楽器では、音の立ち上がりがゆるやかで、そのあと同じ大きさの音が続き、最後は立ち上がりと同じくらいの速さで、音が止まるといった具合だ (図 5.3 参照) 。

これに対して、アナログシンセサイザーと FM 音源では、立ち上がりの速さ (A:アタック) 、その直後の減衰 (D:ディケイ) 、持続の強さ (S:サスティン) 、消える速さ (R:リリース) を調節して、エンベロープを合成する。そのための装置が、**“ADSR”** と呼ばれるものだ。

かつてピンフロイドというロックグループは、“吹けよ風、呼べよ嵐” という曲の中で、シンバルの音を録音したテープを逆に回すことで、少しずつ大きくなって急に止まる効果音を作り出した。でもシンセサイザーを使えば、アタックを遅く、ディケイを速く、サスティンを 0 にするだけで、このような音を合成できる。また、エンベロープの合成は、FM 音源の波形の合成よりも簡単なので、自分で ADSR を調整して効果音を作り出すのも、おもしろいだろう。

図 5.3: 楽器とシンセのエンベロープ

5.1.3 音程が平均律とは限らない

突然だけど、ここでちょっとクラシック音楽の話をする。まずは、音階と周波数の対応関係から整理すると、表 5.2 のようになる。

表 5.2: 音階と周波数の関係

A	440.0Hz
A#	466.2Hz
B	493.9Hz
C	523.3Hz
C#	554.4Hz
D	587.3Hz
D#	622.3Hz
E	659.3Hz
F	698.5Hz
F#	740.0Hz
G	784.0Hz
G#	830.6Hz
a	880.0Hz

このなかで、A の音の周波数は 440Hz 、その 1 オクターブ上の a の音の周波数は 880Hz 。つまり 1 オクターブ離れた音の周波数は、2 倍になっているわけだ。また、半音離れた音の周波数の比は約 1.0595 倍で、12 半音 (1 オクターブ) 離れた音の周波数の比は、1.0595^{12}、つまり 2 になる。

このように、すべての音階の周波数が等比数列で表わされる規則を、“完全平均律”という。この音律は、バロック時代から古典派時代にかけて成立したものらしい。筆者が高校生のころには、“バッハが平均律を作った”と教えられたけど、最近の研究では、バッハよりもあとに平均律ができたという説が有力だそうだ。現代の音楽の大部分は、この平均律に基づいて演奏されている。

さて、この完全平均律ができる以前には、周波数の比が有理数になるような音律が使われていた。これは平均律と異なり、半音の周波数比が場所によって違うもの。たとえばCとC#の比と、BとCの比が違うという特徴があるわけだ。平均律よりも規則が複雑なこともあって、表 5.3 のような各種の音律が設定されている。ただ平均律以外の音律では、和音の響きは美しいけれど、曲の移調が難しいという問題

表 5.3: MSX–Music で設定できる音律一覧

番号	設定される音律	番号	設定される音律
0	ピタゴラス	11	純正律 cis メジャー (b マイナー)
1	ミーントーン	12	純正律 d メジャー (h マイナー)
2	ヴェルクマイスター	13	純正律 es メジャー (c マイナー)
3	ヴェルクマイスター (修正 1)	14	純正律 e メジャー (cis マイナー)
4	ヴェルクマイスター (修正 2)	15	純正律 f メジャー (d マイナー)
5	キルンベルガー	16	純正律 fis メジャー (es マイナー)
6	キルンベルガー (修正)	17	純正律 g メジャー (e マイナー)
7	ヴァロッティ・ヤング	18	純正律 gis メジャー (f マイナー)
8	ラモー	19	純正律 a メジャー (fis マイナー)
9	完全平均律 (初期設定)	20	純正律 b メジャー (g マイナー)
10	純正律 c メジャー (a マイナー)	21	純正律 h メジャー (gis マイナー)

もあるんだ。

バイオリンのように、連続して周波数を変えられる楽器では、どのような音律にも対応できる。でも、ピアノの音律を変えるには、全部の弦をチューニングし直す必要があり、実用的には音律を変えられない。ところが MSX に搭載された FM 音源では、表 5.3 のように音律を選ぶことができる。これにより、純正律のギターのような、簡単には実現できない楽器の音も作れるわけだ。この特徴を利用し、FM 音源のレジスターを操作して音律を微調整すれば、雅楽や琉球音楽なども、精密に演奏することができるかしれない。

なお、FM 音源の周波数は音律が問題になるほど正確だけど、PSG の周波数はそれほどでもない。だから、PSG を楽器の調律や発声練習の基準に使うのは危険だ。とはいっても、筆者は音感がニブイので、音律の違いがよくわからない。FM 音源を使いこなすには、数学、電気、音楽理論の知識に加え、正確な音感とセンスが必要になるけど、そんな人はめったにいない。ゲーム音楽を作るにも、作曲家と音色デザイナーとプログラマーが組んで働くように、役割分担が必要だろう。

5.1.4 MSX–MUSIC を分析してみる

MSX–MUSIC には、あらかじめ 63 種類の音色が用意されている。このうちの 15 種類は FM 音源の LSI に内蔵された音色で、残りの 48 種類は ROM に記録された音色だ。ROM に記録された音色のデータは、

```
CALL VOICE COPY
```

という命令で呼び出すことができる。リスト 5.1 に掲載したのが、このデータを表示するためのプログラムだ。なお、FM 音源に内蔵された音色番号を指定するとエ

ラーが起き、リスト 5.1 のプログラムの場合は、

```
Voice No. * has no data.
```

といったメッセージが表示されるようになっている。

リスト 5.1 (READFM.BAS)

```
100 ' read VOICE DATA of MSX-MUSIC
110 ' by nao-i on 20. Apr. 1990
120 '
130 CALL MUSIC : DEFINT A-Z
140 DIM VI(15),VD(31),VO(3)
150 PRINT "Voice Number (0,...,63; 64 for all; 65 for end) ";
160 INPUT ME
170 IF 0 <= ME AND ME <= 63 THEN VN=ME : GOSUB 200 : GOTO 150
180 IF ME = 64 THEN FOR VN=0 TO 63 : GOSUB 200 : NEXT VN : GOTO 150
190 END
200 '
210 ON ERROR GOTO 460
220 CALL VOICE COPY(@VN,VI)
230 ON ERROR GOTO 0
240 FOR I=0 TO 15
250   VD(I*2)=VI(I) AND 255
260   VD(I*2+1)=(VI(I) / 256) AND 255
270 NEXT I
280 NA$=""
290 FOR I=0 TO 8
300   IF VD(I) THEN NA$=NA$+CHR$(VD(I))
310 NEXT I
320 PRINT : PRINT "Voice No.";VN;" : " NA$
330 PRINT "Transpose=";VI(4);
340 PRINT "  Feedback=";(VD(10) AND 14) / 2
350 FOR I=0 TO 3: VO(I)=VD(I+16): NEXT I
360 PRINT "Operator 0" : GOSUB 490
370 FOR I=0 TO 3: VO(I)=VD(I+24): NEXT I
380 PRINT "Operator 1" : GOSUB 490
390 CALL BGM(0)
400 CALL VOICE(@VN,@VN,@VN)
410 PLAY #2,"CED<G>CR","V6EGF<B>ER","V4GBADGR"
420 CALL VOICE(@0,@0,@0)
430 CALL BGM(1)
440 ON ERROR GOTO 0
450 RETURN
460 '*** error
470 PRINT "Voice No.";VN;" has no datum."
480 RESUME 440
490 '*** print data of an operator
500 PRINT "  AM =";(VO(0) ¥ 128) AND 1;
510 PRINT "  PM =";(VO(0) ¥ 64) AND 1;
520 PRINT "  EG =";(VO(0) ¥ 32) AND 1;
530 PRINT "  KSR=";(VO(0) ¥ 16) AND 1;
```

```
540 PRINT "  MULT=";VO(0) AND 15;
550 PRINT "  LKS=";(VO(1) ¥ 64) AND 3;
560 PRINT "  TL=";VO(1) AND 63
570 PRINT "  ADSR=";(VO(2) ¥ 16) AND 15;","; VO(2) AND 15;",";
580 PRINT (VO(3) ¥ 16) AND 15;","; VO(3) AND 15
590 RETURN
600 ON ERROR GOTO 0
```

プログラム中で操作しているオペレーターは2種類。オペレーター1は、基本の波形を作るための **“キャリアー・オペレーター”** で、オペレーター2が、1を変調するための **“モジュレーター・オペレーター”** だ。それぞれ設定している数値の意味を解説すると、それだけで1冊の本になってしまうので、ここでは省略。参考書などを使って、各自で調べてほしい。そうそう、リスト5.1の“PRINT”という部分を“LPRINT”に変更して、音色データの表を印字しておけば、自分で音色を設計するための参考資料として便利かもしれない。

5.1.5 FM 音源を使ってリズム音に挑戦

MSX–MUSIC に限らず、FM 音源が苦手とする音は打楽器音だ。実際の打楽器や、アナログシンセサイザーが作り出す打楽器音は不規則なノイズだけど、FM 音源の打楽器音は規則性がありすぎることが災いして、“安っぽい”あるいは“機械的な”音となってしまう。

そこで MSX–MUSIC には、63種類の楽器音とはべつに、**“リズム音”** を発生するための機能が用意されている。そもそも63種類の楽器音の中には、打楽器の音色も含まれているのだけれど、これらは楽器音と同じ方法で合成される音色。ここでいうリズム音とは、あくまでも別物だ。

マニュアルなどにも書かれているように、MSX–MUSIC にはチャンネル1から9までの、9組の2オペレーター式 FM 音源が内蔵されている。その全部を楽器音として使えば、9声の演奏が可能なわけだ。

ところが、リズム音を作り出すための方法として、チャンネル1から6までを普通の楽器音に割り当て、チャンネル7から9までの3組分の6オペレーターを、リズム音として使うことも可能になっている。そのため、MSX–MUSIC の機能を表わすのに、“9楽器音または6楽器音+1打楽器音”という表現が使われるわけだ。

BASIC からこれらの機能を利用するには、“CALL MUSIC”命令のパラメーターを変更すればいい。たとえば、

```
CALL MUSIC(0,0,1,1,1,1,1,1,1,1,1)
```

で9楽器音が。

```
CALL MUSIC(1,0,1,1,1,1,1,1)
```

で6楽器音+1打楽器音が選択される。

参考までに次に掲載したプログラムは、MSX–MUSICの音色データと、自分で作り出したリズム音の違いを聞きわけるためのもの。はじめに、楽器音の音色31番の2オペレーターによる“Bass Drum”を4回鳴らしたあと、リズム音の6オペレーターによるバスドラム音を4回鳴らす。リズム音のほうが本物のドラムに似ていることを、実際に打ち込んで、自分の耳で確認してみよう。

リスト 5.2 (BASSDRUM.BAS)

```
10 CALL MUSIC (1,0,1,1,1,3)
20 CALL BGM(0)
30 CALL VOICE(@31)
40 PLAY #2,"V15CCCC","","","","RRRRB!4B!4B!4B!4"
50 CALL VOICE(@0)
```

また、MSXではFM音源とPSGを同時に鳴らせるので、FM音源で楽器音を出し、PSGで打楽器音と効果音を出すことも可能だ。しかし、MSX本体の機種によって、FM音源の音の大きさとPSGの音の大きさのバランスが違っているので、それぞれのマシンに応じて音量を調整するためのプログラムが、必要になってくる。

5.2 FM 音源をコントロール

いまやゲームの BGM や効果音には欠かせない存在となった FM 音源。このページでは、マシン語プログラムから FM 音源をコントロールすることに、挑戦してみよう。

5.2.1 マシン語プログラムで音を出してみる

前のページでも紹介したように、拡張 BASIC を使えば、簡単に FM 音源を操作することができた。でも、ゲームの効果音や BGM として応用するには、マシン語プログラムが直接 **"FM–BIOS"** を呼び出して、FM 音源を操作する必要がある。

ここからは、MSX–C で作られたプログラムが FM 音源を操作するための、ライブラリーとプログラム例を紹介しようと思う。当然のことながら、このプログラムをコンパイルして実行可能なマシン語ファイルにするためには、"MSX–DOS TOOLS (または DOS2 TOOLS)" と、"MSX–C ver.1.1 (または ver.1.2)" が、それぞれ必要になってくる。

さて、リスト 5.3 は、MSX–C で作られたテストプログラムの "TESTFM.C "。"fmdata" という配列がテストデータで、バスドラムとスネアドラムを叩きながら、"ドレミファソラシド" を 4 回演奏させるものだ。このデータの作り方は、アスキーネット MSX にある、msx.spec のボードで公開された資料にも書かれている。

リスト 5.3 (TESTFM.C)

```
/*
 *      testfm.c
 *      by nao-i on 29. May. 1990
 *      (C) Isikawa 1990
 *      free to use and copy, but no guarantee or support
 */
#include <stdio.h>
#include "fmlib.h"
#pragma nonrec

#define TESTLENGTH      20
#define TESTTIMES       4

static  char    fmdata[] = {            /* test data            */
        14, 0,                          /* 0: offset to rythme  */
        33, 0,                          /* 2: offset to ch 1    */
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,   /* 4: offset to ch 2...6*/

        FM_RVOL + 31, 8,                /* 14: rhythm VOL = 8   */
        0x30, TESTLENGTH,               /* 16: B drum           */
        0x28, TESTLENGTH,               /* 18: S drum           */
```

```
        0x28, TESTLENGTH,               /* 20: S drum         */
        0x28, TESTLENGTH,               /* 22: S drum         */
        0x30, TESTLENGTH,               /* 24: B drum         */
        0x28, TESTLENGTH,               /* 26: S drum         */
        0x28, TESTLENGTH,               /* 28: S drum         */
        0x28, TESTLENGTH,               /* 30: S drum         */
        FM_END,                         /* 32: end of rhythm  */

        FM_VOL + 8,                     /* 33: Ch. 1 VOL = 8  */
        FM_INST + 3,                    /* 34: Guiter         */
        FM_SUSON,
        FM_LEGOFF,
        FM_Q, 6,
        FM_O4 + FM_C, TESTLENGTH,
        FM_O4 + FM_D, TESTLENGTH,
        FM_O4 + FM_E, TESTLENGTH,
        FM_O4 + FM_F, TESTLENGTH,
        FM_O4 + FM_G, TESTLENGTH,
        FM_O4 + FM_A, TESTLENGTH,
        FM_O4 + FM_B, TESTLENGTH,
        FM_O5 + FM_C, TESTLENGTH,
        FM_END                          /* end of Ch. 1       */
        };


VOID    main(argc, argv)
int     argc;
char    **argv;
{
        auto char fmwork[FMWORK]; /* address must be >= 8000H */
        auto char fmbuf[256];     /* address must be >= 8000H */
        char      fmstat;

        if ((fmstat = fmopen(fmwork)) == 1) {
                puts("No FM-BIOS.");
                exit(1);
        } else if (fmstat == 2) {
                puts("Bad address.");
                exit(1);
        }
        printf("fmopen : address of work area is %04X¥n",
          (unsigned)fmwork);
        memcpy(fmbuf, fmdata, sizeof(fmdata));
        fmstart(fmbuf, (char)TESTTIMES);
        do {
                fputs("Playing.¥015", stdout);
        } while (fmtest());
        fputs("¥nEnd of play.¥n", stdout);
        fmstop();
        fputs("fmstop : complete¥n", stdout);
        fmclose();
        fputs("fmclose : complete¥n", stdout);
        exit(0);
}
```

それでは、プログラムを簡単に説明していこう。まずは、大きさが “FMWORK” バイトの auto 配列である “fmwork” を用意する。そして、その番地をわたして、ライブラリーの “fmopen” を呼び出す。この配列は 8000H 以上の番地に置かれる必要があるので、static ではなく auto と宣言しよう。

以上の手続きにより、FM 音源の準備に成功すれば 0 が、FM 音源がなければ 1 が、“fmwork” の番地が 7FFFH 以下ならば 2 が、それぞれ “fmopen” から返されるはずだ。

このとき注意しなければいけないのは、FM 音源を使う前にかならず “fmopen” を呼び出し、プログラムが終了する前に “fmclose” を呼び出す必要があるということ。もしも “fmclose” が呼ばれる前に、プログラムが終了してしまうと都合が悪いので、このライブラリーでは [CTRL]+[C] キーや、[CTRL]+[STOP] キーが押されても、無視するようになっている。

もしも FM 音源を使ったプログラムを自作しようなんてときは、[CTRL]+[C] キーや、ディスクエラーに対する処理をきちんとすることが大切だ。どんな場合でも、“fmclose” を呼び出してから、終了させるように注意しよう。

さて、データが入った番地と、演奏回数のパラメーターをわたして “fmstart” を呼び出すと、すぐに FM–BIOS が演奏をはじめる。この FM–BIOS は、タイマー割り込みで動くようになっているので、演奏を続けながらもプログラムを先に進めることも可能だ。このプログラムでは、演奏しながら画面に “Playing.” と表示させるようにしてみた。

“fmtest” は、演奏中ならば 1 を、演奏が終わっていれば 0 を返す。また “fmstop” は、演奏を終了させて、FM–BIOS を初期化するためのもの。

5.2.2 ライブラリーの概要を説明する

リスト 5.4 に掲載したのは、FM 音源ライブラリーの関数と定数を定義するための、ヘッダーファイル “FMLIB.H” 。

リスト 5.4 (FMLIB.H)

```
/*
 *      fmlib.h : header file for fmlib
 *      by nao-i on 31. May. 1990
 * by nao-i on 24. Feb. 1991  FM_01 changed from 0 to 1
 *      (C) Isikawa 1990
 *      free to use and copy, but no guarantee or support
 */
extern  char    fmopen();       /* please call this first       */
extern  VOID    fmclose();      /* please call this last        */
```

```
extern  VOID    fmwrite();      /* write to OPLL register       */
extern  VOID    fmotir();       /* write to OPLL register 0...7 */
extern  VOID    fmstart();      /* back ground music            */
extern  VOID    fmstop();       /* stop back ground music       */
extern  char    *fmread();      /* read data from ROM           */
extern  char    fmtest();       /* now playing ?                */
#define FMWORK  (0x00a0+32)     /* size of work area            */

#define FM_VOL          0x0060  /* volume 60H...6FH             */
#define FM_INST         0x0070  /* instulment 70H...7FH         */
#define FM_SUSOFF       0x0080  /* sustain off                  */
#define FM_SUSON        0x0081  /* sustain on                   */
#define FM_EXPINST      0x0082  /* expandet instulment          */
#define FM_USRINST      0x0083  /* user-defined instulment      */
#define FM_LEGOFF       0x0084  /* legato off                   */
#define FM_LEGON        0x0085  /* legato on                    */
#define FM_Q            0x0086  /* Q                            */
#define FM_END          0x00ff  /* end of data                  */
#define FM_RVOL         0x00a0  /* volume of rhythm             */

/* pitch */
#define FM_C    0       /* C    */
#define FM_CS   1       /* C#   */
#define FM_D    2
#define FM_DS   3
#define FM_E    4
#define FM_F    5
#define FM_FS   5
#define FM_G    7
#define FM_GS   8
#define FM_A    9
#define FM_AS   10
#define FM_B    11
/* octove */
#define FM_01   1       /* FM_01+FM_C means C of octove 0       */
#define FM_02   13
#define FM_03   25
#define FM_04   37
#define FM_05   49
#define FM_06   61
#define FM_07   73
#define FM_08   85
```

そして、次の長大なリスト 5.5 が、FM 音源ライブラリーだ。リストのはじめから順番に、BIOS などの番地の定義、FM–BIOS を呼び出すマクロの定義、ライブラリーが使用するワークエリアの定義、そしてライブラリーのプログラム本体が書かれている。

リスト 5.5 (FMLIB.Z80)

```
;
;       fmlib.z80 : library for MSX-C
;       by nao-i on 29. May. 1990
;       (C) ASCII 1988 for 'search', (C) Isikawa 1990
;       free to use and copy, but no guarantee or support
;
;       .Z80
;
;       address of BIOS and system work area
;
rdslt   equ     000ch
calslt  equ     001ch
enaslt  equ     0024h
breakv  equ     0f325h          ; ^C break vector
ramad0  equ     0f341h          ; slot # of RAM
ramad1  equ     0f342h
ramad2  equ     0f343h
ramad3  equ     0f344h
exptbl  equ     0fcc1h
h.timi  equ     0fd9fh          ; timer interrupt hook
;
;       address of FM-BIOS jump table
;
ids[illegible]tr  equ     4018h+4
_wr[illegible]to  equ     4110h
_ir[illegible]io  equ     4113h
_ms[illegible]ta  equ     4116h
_ms[illegible]to  equ     4119h
_rd[illegible]da  equ     411ch
_op[illegible]ld  equ     411fh
_ts[illegible]tb  equ     4122h
;
;       MACROs to call FM-BIOS
;
CALLFM  MACRO   ADDRESS
        ld      ix,address
        ld      iy,(biosslot-1)
        call    calslt
        ENDM
;
JUMPFM  MACRO   ADDRESS
        ld      ix,address
        ld      iy,(biosslot-1)
        jp      calslt
        ENDM
;
        dseg
biosslot: ds    1               ; slot of FM-BIOS
breakv:   ds    2               ; saving ^C vector
p.ontime: ds    2               ; address of interrupt handler
p oldhook:ds    2               ; address of saved hook
;
```

```
        cseg
;
ontime:
        push    af
        ld      ix,_opldrv
        ld      iy,0            ; will be modifyed
ontime.slot equ $-1
        call    calslt
        pop     af
oldhook:
        nop
        nop
        nop
        nop
        nop
        ret
sizeof_ontime equ $-ontime
IF      sizeof_ontime GT 31
        .PRINTX "ontime routine too big"
ENDIF
;
hooktbl:
        rst     30h
        db      0               ; will be modifyed
        dw      0               ; will be modifyed
toret:
        ret
vvv:
        dw      toret           ; to ignore ^C
;
;       char    fmopen(address)
;       char    *address;       /* address of work area */
;
;       0 : successful
;       1 : no FM-BIOS
;       2 : bad address of work area
;
fmopen@::
        ld      a,2
        bit     7,h
        ret     z               ; address of work area < 8000H

        ld      (p.ontime),hl
        push    hl
        ex      de,hl
        ld      hl,ontime
        ld      bc,sizeof_ontime
        ldir                    ; trasfer ontime routine
        pop     hl
        push    hl
        ld      de,oldhook-ontime
        add     hl,de
        ld      (p.oldhook),hl
        pop     hl
        ld      de,32
```

```
        add     hl,de
        res     0,l             ; make sure address is even

        push    hl
        call    search
        ld      a,(biosslot)
        ld      hl,(p.ontime)
        ld      de,ontime.slot-ontime
        add     hl,de
        ld      (hl),a          ; modify LD IY,??00
        or      a
        ld      a,1
        pop     hl              ; address of work area
        ret     z               ; no FM-BIOS
        CALLFM  _iniopl
        ld      h,40h
        ld      a,(ramad1)      ; because FM-BIOS
        call    enaslt          ; does not restore slot1
;
        ld      hl,h.timi
        ld      de,(p.oldhook)
        ld      bc,5
        ldir                    ; save h.timi
        ld      hl,hooktbl
        ld      de,h.timi
        ld      bc,5
        di
        ldir                    ; set h.timi
        ld      a,(ramad2)
        ld      (h.timi+1),a
        ld      hl,(p.ontime)
        ld      (h.timi+2),hl
;
        ld      hl,(breakv)
        ld      (breaks),hl     ; save break vector
        ld      hl,vvv
        ld      (breakv),hl     ; set break vector
        ei
        xor     a
        ret
;
;       VOID    fmclose()
;
fmclose@::
        di
        ld      hl,(breaks)
        ld      (breakv),hl     ; restore break vector
        ld      hl,breaks
        ld      hl,(p.oldhook)
        ld      de,h.timi
        ld      bc,5
        ldir                    ; restore h.timi
        ei
        ret
;
```

```
;       VOID    fmwrite(RegNum, Datum)
;       char    RegNum;         /* OPLL register number         */
;       char    Datum;          /* datum to write               */
;
fmwrite@::
        JUMPFM  _wrtopl
;
;       void    fmotir(aData)
;       char    aData[8];
;
fmotir@::
        ld      b,8
        xor     a
        di
fmotir_loop:
        ld      e,(hl)
        inc     hl
        CALLFM  _wrtopl
        inc     a
        djnz    fmotir_loop
        ret
;
;       void    fmstart(pDatum, Times)
;       char    *pData;         /* pointer to Music data        */
;       char    Times;          /* times to play                */
;
fmstart@::
        ld      a,e
        inc     a
        ret     z               ; make sure that Times != 255
        dec     a
        bit     7,h
        ret     z               ; make sure that pData >= 8000H
        JUMPFM  _mstart
;
;       VOID    fmstop()
;
fmstop@::
        JUMPFM  _mstop
;
;       char    *fmread(ptr, num)
;       char    *ptr;
;       char    num;
;
fmread@::
        ld      a,e
        JUMPFM  _rddata
;
;       char    fmtest()
;
fmtest@::
        JUMPFM  _tstbgm
;
; Search FM-BIOS
;
```

```
search:
        ld      b,4
search_id:
        push    bc              ;save counter
        ld      a,4
        sub     b               ;make primary slot number
        ld      c,a             ;save it
        ld      hl,exptbl       ;point expand table
        ld      e,a
        ld      d,0
        add     hl,de
        ld      a,(hl)          ;get the slot is expanded or not
        add     a,a             ;expanded ?
        jr      nc,no_expanded  ;no..
        ld      b,4             ;number of expanded slots
search_exp:
        push    bc              ;save it
        ld      a,24h           ;[a]=00100100b
        sub     b               ;make secondary slot # A=001000ss
        rlca                    ;[a]=01000ss0b
        rlca                    ;[a]=1000ss00b
        or      c               ;make slot address A=1000sspp
        call    chkids          ;check id string
        pop     bc              ;restore counter
        jr      z,search_found  ;exit this loop if found
        djnz    search_exp
not_found:
        xor     a
        ld      (biosslot),a
        pop     bc
        djnz    search_id
        ret

no_expanded:
        ld      a,c             ;get slot address
        call    chkids          ;check id string
        jr      nz,not_found    ;exit this loop is found
search_found:
        pop     bc
        ret
;
id_string:
        db      'OPLL'
id_string_len   equ     $-id_string
;
; Check ID srting
;  Entry : [A]=slot address to check
;  Return: Zero flag is set if ID is found
;  Modify: [AF],[DE],[HL]
;
chkids:
        ld      (biosslot),a
        push    bc              ;save environment
        ld      hl,idstrg
        ld      de,id_string
```

```
        ld      b,id_string_len
chkids_loop:
        push    af              ;save slot address
        push    bc              ;save counter
        push    de              ;save pointer to string
        call    rdslt           ;read a byte
        ei                      ;leave critical
        pop     de              ;restore pointer
        pop     bc              ;restore counter
        ld      c,a             ;save data
        ld      a,(de)          ;get character
        cp      c               ;same ?
        jr      nz,differ       ;no..
        pop     af              ;restore slot address
        inc     de              ;point next
        inc     hl
        djnz    chkids_loop
        pop     bc              ;restore environment
        xor     a               ;found set zero flag
        ret
;
differ:
        pop     af              ;restore slot address
        pop     bc              ;restore environment
        xor     a               ;clear zero flag
        inc     a
        ret
;
        end
```

プログラムのポイントを解説すると、まず “fmopen@” は、“ontime” からのタイマー割り込み処理プログラムをべつの番地に転送し、FM–BIOS が置かれているスロットを探し、初期化し、タイマー割り込みフックを設定するためのもの。割り込み処理プログラムと、それが参照するデータは、8000H 番地以上に置かれる必要があるので、プログラムを転送して、“ld iy, 0” という部分を、“ld iy, FM–BIOS のスロット番号*256” に書き替えている。

FM–BIOS が置かれているスロットを探すプログラム自体は、アスキーネット MSX に公開されている、FM–BIOS の仕様書から引用した。全部のスロットについて、401CH 番地に “OPLL” という文字列があるかを調べることで、FM–BIOS を探している。

さて、“fmopen@” にわたされた 192 バイトのワークエリアは、割り込み処理プログラム、“h.timi” の元の内容の保存場所、FM–BIOS のワークエリア (160 バイト) に使われる。このとき FM–BIOS のワークエリアは、偶数番地からはじまる必要があるので、余分にワークエリアを用意し開始番地を切り上げている。

これは仕様書に書かれてなかったのだけど、“CALLFM _iniopl” で FM–BIOS を初期化すると、ページ 1 がべつのスロットに切り替えられたまま戻ってくることがあった。かならず、ページ 1 を元のスロットに戻す必要がある。

前にも書いたように、タイマー割り込みフックを書き替えたままプログラムが終了すると困るので、MSX–DOS のワークエリアの F325H 番地を書き替えて、CTRL+C キーを無視するようにした。もしプログラムがディスクを使うならば、ディスクエラーを処理するプログラムを追加する必要がある。そして “fmclose@” は、タイマー割り込みフックとCTRL+C キーの処理を元に戻すためのものだ。

ライブラリーの残りの部分については、レジスターに必要な値を入れて、FM–BIOS を呼び出すだけで使用することができる。各自でいろいろ試してみよう。

5.2.3 MSX–C でコンパイルしよう

MSX–DOS の MED や KID などのエディターでリストを打ち込んだら、リスト 5.3 を “TESTFM.C” 、リスト 5.4 を “FMLIB.H” 、そしてリスト 5.5 を “FMLIB.Z80” というファイル名でセーブしよう。次の手順でコンパイルすると “TESTFM.COM” ができるはずだ。プログラムを実行するには、MSX–DOS のコマンドラインから “TESTFM ↵” と入力すればいい。

リスト 5.6 (FMLIB.BAT)

```
m80 ,=fmlib.z80/r/m/z
cf testfm
cg testfm
m80 ,=testfm.mac/r/m/z
l80 testfm,fmlib,ck,clib/s,crun/s,cend,testfm/n/y/e:xmain
```

5.3 FM音源のデータ構造だ

このページでは、FM音源のデータ構造と、実際に音源データを指定する方法を説明する。前に説明したマシン語でFM音源を操作するプログラムと合わせて利用しよう。

5.3.1 FM音源のデータを作ってみよう

前のページでは、FM–BIOSをマシン語から呼び出して、音を出すためのプログラム例を紹介した。ここでは、そのプログラムを使って演奏するための、FM音源のデータの作り方を説明しよう。

FM–BIOSのデータ構造を要約するなら、大きな配列ということになる。配列の中にあるそれぞれのデータ位置は、配列の先頭からのバイト数で数えられ、“オフセット”と呼ばれている。ただし、配列の先頭のオフセットは0だ。また、配列の先頭は“0バイト目”、その次は“1バイト目”というように、呼ばれることもある。

表5.4に掲載したのは、6楽器音+1打楽器音のデータ構造と、実際のデータ例を示したもの。これを見てもらえばわかるように、データ本体は配列の末尾の方に並べられ、配列の先頭の14バイトには、それぞれのデータが置かれるオフセットが書き込まれている。

表5.4: 6楽器音+1打楽器音のデータ構造

オフセット	内容
0	打楽器音データのオフセット (注1)
2	楽器音1データのオフセット (注2)
4	楽器音2データのオフセット (注2)
6	楽器音3データのオフセット (注2)
8	楽器音4データのオフセット (注2)
10	楽器音5データのオフセット (注2)
12	楽器音6データのオフセット (注2)
14	打楽器音データ
(注3)	楽器音1データ
(注3)	楽器音6データ

(注1) かならず0EH、00Hの2バイトを書き込む。
(注2) 楽器音データの開始位置の、この表のデータの先頭からのバイト単位のオフセットを、それぞれ下位バイト、上位バイトの順に指定する。また、使わないチャンネルに対しては0を指定する。
(注3) オフセットで指定された場所に楽器音データが置かれる。

順番にデータ構造を説明していくと、まず0バイト目と1バイト目(オフセット0と1)は、打楽器音データが置かれているオフセットで、かならず14が書き込まれる。この値をCPUのZ80が理解できるように、下位バイト、上位バイトの順に2バイトで表わすと、それぞれ0EH、00Hとなる。

続く2バイト目から13バイト目(オフセット2~13)には、楽器音のチャンネル1~6までのデータのオフセットが置かれる。もしもこのときに、オフセット0が指

表 5.5: 6 楽器音+1 打楽器音のデータ構造の例

オフセット	内容
0	0EH 00H
2	21H 00H
4	00H 00H
6	00H 00H
8	00H 00H
10	00H 00H
12	00H 00H
14~	打楽器音データ
33~	楽器音 1 データ

これは 6 楽器音+1 打楽器音のデータ構造の例。14~32 バイト目までに打楽器音の音源データが、32 バイト目からは楽器音の第 1 チャンネルの音源データが置かれ、楽器音の第 2~6 チャンネルは使われていない。

定されれば、そのチャンネルは使われないというわけだ。

たとえば表 5.5 のデータ構造例では、2 バイト目と 3 バイト目 (オフセット 2 と 3) に、21H と 00H が書かれている。これが意味するのは、楽器音のチャンネル 1 の音源データが、オフセット 33 以降に置かれているということだ。そして 4 バイト目から 13 バイト目 (オフセット 4~13) には、00H が書き込まれているので、楽器音のチャンネル 2~6 は使われないことがわかる。

前に紹介した FM 音源をコントロールするプログラムでは、C 言語のプログラムの中にテストデータを埋め込んだので、データの長さを数えてオフセットを指定する必要があった。しかし、よく考えると、以下のようなアセンブラーのプログラムで、音源データを作るほうが簡単だったかもしれない。参考までに書いておくと、

```
FMDATA:
    DW 14
    DW CH1-FMDATA
    DW CH2-FMDATA
    DW CH3-FMDATA
    DW CH4-FMDATA
    DW CH5-FMDATA
    DW CH6-FMDATA
    DB 打楽器音データ...
CH1:
    DB チャンネル 1 データ...
CH2:
    DB チャンネル 2 データ...
(以下、CH6 まで同様)
```

ということになる。このプログラムなら、ソースリストをアセンブルするときに、MSX-DOS のアセンブラー (M80) がオフセットを自動的に計算してくれるからだ。

もっとも、このプログラムをそのまま実用として使うことはできない。BASIC 言語における PLAY 文の役割を果たすような、ミュージック・マクロ・ランゲージ

(MML) を、FM–BIOS の音源データに変換するためのプログラムが、必要になるだろう。

さて、打楽器音なしで 9 楽器音を演奏する場合は、表 5.6 に掲載したようなデータ構造になる。基本的には、6 楽器音+1 打楽器音のデータ構造と同じで、まず 0 バイト目から 17 バイト目 (オフセット 0〜17) までに、楽器音のチャンネル 1〜9 の音源データが書き込まれたオフセットを指定する。もしも 00H が指定された場合は、該当するチャンネルは使われないということだ。

また、チャンネル 1 の音源データは、かならず 18 バイト目 (オフセット 18) からはじまることになるので、チャンネル 1 のデータのオフセットには、12H 、00H (10 進数で 18) の値がいつでも書き込まれている。

表 5.6: 9 楽器音のデータ構造

オフセット	内容
0	楽器音 1 データのオフセット (注 1)
2	楽器音 2 データのオフセット (注 2)
⋮	
16	楽器音 9 データのオフセット (注 2)
18	楽器音 1 データ
⋮	
(注 3)	楽器音 9 データ

(注 1) かならず 12H 、00H の 2 バイトを書き込む。
(注 2) 楽器音データの開始位置の、この表のデータの先頭からのバイト単位のオフセットを、それぞれ下位バイト、上位バイトの順に指定する。また、使わないチャンネルに対しては 0 を指定する。
(注 3) オフセットで指定された場所に楽器音データが置かれる。

なお、これは余談になるけど、FM–BIOS は、音源データの先頭が 0EH であるか 12H であるかによって、打楽器音の有無 (つまり 6 楽器音+1 打楽器音か、9 楽器音のみか) を決めている。だから、打楽器音のデータはかならず 14 バイト目 (オフセット 14) から、打楽器音がない場合にはチャンネル 1 の楽器音データがかならず 18 バイト目 (オフセット 18) から、はじまる必要があるようだ。

5.3.2 打楽器音のデータを指定するには

図 5.4 に掲載したのが、打楽器音データの詳細と、実際のデータ列だ。ここでは、5 種類の打楽器を “BSTCH” のアルファベットで、音のデータを 2 進数でそれぞれ表わしている。

まずは次のような 2 バイトのデータで、打楽器ごとの音量を指定してみよう。

```
101BSTCH
0000VVVV
```

図 5.4: 打楽器音のデータ

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	B	S	T	C	H
0	0	0	0	音量 (0~15)			
0	0	1	B	S	T	C	H
音長 (1~255)							

B　バスドラム
S　スネアドラム
T　タム
C　シンバル
H　ハイハット

```
10110000    バスドラム
00000000    音量 0
10101000    スネアドラム
00000001    音量 1
00110000    バスドラム
00010100    音長 20
00101000    スネアドラム
00010100    音長 20
00101000    スネアドラム
00010100    音長 20
00101000    スネアドラム
00010100    音長 20
11111111    データ終了
```

打楽器を選択するには、BSTCH と書かれた 5 ビットについて、それぞれ 1 か 0 を指定する。音量は最大音量に対する減衰量を、音長は音を出してから次の音を出すまでの間隔を表わしている。

このとき、“BSTCH” の 5 ビットの列には、音量を指定したい打楽器を 1 、そうでないものを 0 で指定する。また、“VVVV” の部分には、音量を指定する 0~15 の値 (実際には 2 進数の値) が入る。ただし、このときの “音量” は、最大音量に対する減衰量を指定するので、0 なら最大の音が、15 なら最小の音が出るので注意しよう。

たとえば、バスドラムの音量を 0 、スネアドラムとシンバルの音量を 1 に設定するには、

```
10110000
00000000
10101010
00000001
```

と指定すればいい。

音量の指定が終わったら、次に打楽器ごとの音長を指定しよう。ただし、打楽器音自体の長さは常に一定なので、この場合の “音長” とは、音を出してから次の音を出すまでの間隔を指している。そして、

```
001BSTCH
```

によって、打楽器の種類が指定され、次の 1 バイトで音長 (255 までの値) を指定する。255 以上の音長を指定するには、まず 255 を書き、その次に実際の音長から

255を引いた値を指定する。この値が255以上ならば、同様の操作を繰り返せばいい。たとえば、

```
00110000
11111111
00000000
```

はバスドラム、音長255を表わし、

```
00110010
11111111
11111111
11111111
11101011
```

は、バスドラムとシンバル、音長1000を表わしている。

FM音源の仕様書には“音長”の単位が書かれていなかったけど、テストデータで実測した結果、音長の単位はタイマー割り込みの周期と同じ、60分の1秒だった。

5.3.3　楽器音のデータを指定してみよう

表5.7に掲載したのが、楽器音のデータの詳細だ。このデータの中には、表の1バイトの値だけで意味を持つものと、続く1バイトまたは2バイトの値との組み合わせで、意味を持つものがある。

実際に楽器音データを指定する順番は、音量、音色、サスティン、レガート、Qの順だ。

音量の指定は打楽器のデータと同じで、最大音量からの減衰量で表わす。つまり、0で最大の音が、15で最小の音が出るという具合。

サスティンは、楽器音の減衰を調整するためのものだ。楽器音のエンベロープは前にも説明したように、“ADSR”という値で決定される。繰り返すなら、Aはアタック(立ち上がりの速さ)、Dはディケイ(減衰)、Sがサスティン(持続の強さ)、Rがリリース(消える速さ)の略だ。

OPLL内蔵音色の“ADSR”は、音色ごとに固定されている。でも、サスティンをオンにすると、リリースが遅くなって、音が伸びる。さらに、サスティンはチャンネルべつに指定可能なので、チャンネル1と2の両方にギター音を割り当て、チャンネル1だけのサスティンをオンにするというような、細かい工夫も可能だ。

レガートをオンにすると、ひとつの音符と次の音符との音がつながる。ただし、レガートを使いすぎると曲のメリハリがなくなるので、一部分のチャンネルのみのレガートをオンにするような工夫が必要だろう。

表 5.7: 楽器音のデータ

値	意味
00H	休符、続く 1 バイトが音長
01H	オクターブ 1 の C 、続く 1 バイトが音長
⋮	⋮
5FH	オクターブ 7 の A#、続く 1 バイトが音長
60H	音量 0 (最大音量)
⋮	⋮
6FH	音量 15 (最小音量)
70H	音色 0 (ROM 内蔵音色またはユーザー指定音色)
71H	音色 1 (バイオリン)
⋮	⋮
7FH	音色 15 (エレキベース)
80H	サスティン・オフ
81H	サスティン・オン
82H	続く 1 バイト (00H~3FH) が ROM 内蔵音色番号
83H	続く 2 バイト (下位、上位の順) が音色データの番地
84H	レガート・オフ
85H	レガート・オン
86H	続く 1 バイト (01H~08H) が Q
FFH	データ終了

Q に指定できる値は 1~8 で、音符の長さと実際に音を出す長さの比を表わしている。たとえば、$Q = 6$ で音符の長さが 80 ならば、$80 \times 6 \div 8 = 60$ の長さの音が出て、$80 \times (8 - 6) \div 8 = 20$ の休みが入る。

以上の、レガート、サスティン、Q に指定する値の組み合わせで、音符のつながり方、つまり曲の滑らかさが決まるわけだ。

表 5.8: 楽器音のデータの例

データ	意味
68H	音量 8
73H	音色 3(ギター)
81H	サスティン・オン
84H	レガート・オフ
86H, 06H	$Q = 6$
25H, 14H	オクターブ 4 の C 、音長 20
27H, 14H	オクターブ 4 の D 、音長 20
29H, FFH, FFH, FFH, EBH	オクターブ 4 の E 、音長 1000
FFH	データ終了

次に、音符ごとの音階と音長を指定する。00H～5FHまでの1バイトの値が音階を表わし、その次の1バイトが音長を表わしている。たとえば、

```
25H,14H
```

の2バイトのデータは、それぞれオクターブ4のCと、音長20を表わしている。また、255以上の音長を表わす方法は打楽器の場合と同様で、たとえば、

```
29H,FFH,FFH,FFH,EBH
```

の5バイトのデータは、オクターブ4のEと、音長1000を表わしているわけだ。

5.3.4 OPLLドライバーでできないこと

FM–BIOSが持つ機能の中で、タイマー割り込みにより呼び出されて与えられたデータを、自動的に演奏するものを、**“OPLLドライバー”**と呼ぶ。ここでは、このドライバーを利用して、BASICの

```
CALL PITCH
CALL TEMPER
CALL TRANSPOSE
```

と同じ機能を実現する方法を、解説するつもりでいた。ところが、FM–BIOSの開発元に問い合わせてみたところが、“自分でやってください”とのこと。つまり、自分でOPLLのレジスターを書き替えないと、できないことが判明してしまったのだ。

結局、OPLLドライバーを使った場合は、12平均率でAが440ヘルツの標準的な音律でのみ、演奏が可能だということ。BASICがサポートするMMLには、音量を細かく設定する機能と、音源チップのレジスターに値を書き込む機能があるけど、FM–BIOSのドライバーで同じことをするのは不可能だ。また、ゲームのバックグラウンドに音楽を鳴らしながら、効果音を出すことも困難だ。

こうして考えてみると、FM音源を使いやすいものにするには、いま述べたような機能を追加したドライバーと、MMLをそのドライバーのデータに変換するプログラムが必要になりそうだ。ここまでの記事と、あとで紹介するFM音源に関する参考書があれば、必要な情報はそろうはず。プログラムに自信のある人は、ぜひとも、挑戦してみよう。

5.3.5 音色データを追加してみよう

前にも書いたように、FM音源のLSI (OPLL)には、15種類の、それをコントロールするFM–BIOSのROMには、48種類の音色データが用意されている。けれ

図 5.5: 音色データ

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
0	AM	VIB	EGT	KSR	Multiple				Op.0
1	AM	VIB	EGT	KSR	Multiple				Op.1
2	KSL (Op.0)		Total LEVEL MODELATER						
3	KSL (Op.1)		空き	DC	DM	Feedback			
4	Attack (Op.0)				Decay (Op.0)				
5	Attack (Op.1)				Decay (Op.1)				
6	Sustain (Op.0)				Release (Op.0)				
7	Sustain (Op.1)				Release (Op.1)				

Op.0 はモジュレーター・オペレーター、Op.1 はキャリアー・オペレーターを表わす。この 8 バイトが、OPLL のレジスター 0~7 に書き込まれる。詳細は "MSX2+パワフル活用法 (アスキー刊)" を参照。

ども、図 5.5 に掲載したようなデータ構造で、さらに音色を追加することもできるようになっている。

この 8 バイトの音色データは、FM 音源の LSI (OPLL) のレジスター 0~7 に、そのまま書き込まれる。BASIC の拡張コマンドである、

```
CALL VOICE
CALL VOICE COPY
```

に使われる 32 バイトのデータと、形式が異なることに注意しよう。

OPLL の内部では、各チャンネルごとの音色が、0~15 の値で指定されている。このとき、音色 0 は OPLL のレジスター 0~7 までで設定される、オリジナル音色を表わしている。つまり、自作の音色データまたは、FM–BIOS の ROM に内蔵された音色データは、同時に 1 種類のみ使えるわけだ。

このことは、音色の数の制限により発生したこと。チャンネル数の制限ではない。だから、たとえばチャンネル 1 と 2 に自作の音色を割り当て、チャンネル 2~4 に OPLL に内蔵された音色を、割り当てることも可能なのだ。

音色データの内容を解説していると、それだけで 1 冊の本が書けてしまうので、今回は割愛。そのかわり、参考書を紹介する。

"MSX2+ パワフル活用法" 杉谷成一著・アスキー出版局刊

ただし、この本には若干の間違いがあるようだ。あとのページに内容訂正を掲載しておくので、各自で修正しておこう。

このほか、パソコン通信をしているなら一度アクセスしてほしいのが、アスキーネット MSX の "msx.spec" というボード。ここには、マシン語のプログラムが MSX–

MUSICを使うための、“FM–BIOS”の仕様書が公開されている。また、それだけでなく、MSXに関するさまざまな情報が掲載されているのだ。

5.3.6 サンプルデータを解説する

それでは最後にまとめとして、実際にデータを指定した例を紹介しよう。リスト5.7は、前にも掲載したプログラムリストの一部分だ。

まず一番上の部分では、チャンネルごとのデータのオフセットを指定している。打楽器音のデータが14バイト目から、楽器音チャンネル1のデータが33バイト目からはじまり、チャンネル2~5は使われないことを表わしている。

リストの中央の部分は、打楽器音のデータだ。まず、全部の打楽器の音量を8に設定している。“FM_RVOL”の値は0xa0で、これに31を加えると0xbf、つまり全打楽器の音量指定になるわけだ。引き続きバスドラムを音長20で、スネアドラムも音長20でというように、順番に音を指定していく。“FM_END”の値は0xffで、データの終わりを表わしている。

リストの下半分は、楽器音チャンネル1のデータだ。まず音量8、音色3 (OPLL内蔵のギター)、サスティン・オン、レガート・オフ、Q=6を指定する。そして、“ドレミファソラシド”を各音長20で鳴らし、“FM_END”まできたら終了。このとき、音階を直接0~95までの数値で指定すると不便なので、12音階とオクターブの値をわけている。たとえば、

```
FM_O4 + FM_D
```

によって、オクターブ4のDを指定するわけだ。“FM_O4”の値は37で、“FM_D”の値は2だから、これらを足すとオクターブ4のDを表わす39になる。

なお、アセンブラー(M80)でテストデータを作るためには、次のように定数を定義すると便利だろう。

```
FM_C    EQU 1
FM_CS   EQU 2
        ⋮
FM_VOL  EQU 60H
        ⋮
```

そして音量を、

```
DB FM_VOL + 8
```

のように。同じく音階を、

```
DB FM_04 + FM_C
```

のように指定すればいい。

リスト 5.7 (楽器音のサンプルデータ)

```
#define TESTLENGTH      20
#define TESTTIMES       4

static  char    fmdata[] = {
/* ここでは、各チャンネルごとのオフセットを指定している。 */
        14, 0,
        33, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

/* これは打楽器音のデータ。                 */
/* はじめに全体の音量を設定したあとで、     */
/* それぞれの楽器の音長指定していく。       */
        FM_RVOL + 31, 8,
        0x30, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        0x30, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        FM_END,

/* これは、楽器音チャンネル１のデータ。                  */
/* このリストでは、チャンネル２～５は使われていない。 */
        FM_VOL + 8,
        FM_INST + 3,
        FM_SUSON,
        FM_LEGOFF,
        FM_Q, 6,
        FM_04 + FM_C, TESTLENGTH,
        FM_04 + FM_D, TESTLENGTH,
        FM_04 + FM_E, TESTLENGTH,
        FM_04 + FM_F, TESTLENGTH,
        FM_04 + FM_G, TESTLENGTH,
        FM_04 + FM_A, TESTLENGTH,
        FM_04 + FM_B, TESTLENGTH,
        FM_05 + FM_C, TESTLENGTH,
        FM_END
        };
```

5.4 FM 音源にまつわるアレコレ

ここまでの説明で、FM 音源に関する説明はすべて終わったと思っていたら、やり残していたことが出てきてしまった。もうしばらく、おつきあいください。

5.4.1 パワフル活用法の内容訂正

MSX2+マシンの参考書として紹介してきた "MSX2+ パワフル活用法"(アスキー刊、価格 1240 円 [税込]) に、いくつかの誤りが見つかった。本に書いてあるとおりにプログラムしても、予定どおりに動かないで頭をかかえている人も多いはず。このページでは、現在わかっている範囲での誤りを、訂正していくことにする。もしも、ここに記載した以外にも誤りを見つけた人がいたら、M マガ編集部あてに、ぜひ知らせてください。

それでは、まず、ひとつ目の訂正から。147 ページに掲載されている、"表 4.4 音色ライブラリー一覧表" を見てみよう。この中の音色番号 10 、音色名が "Guitar" となっている項目の "OPLL VOICE" の欄に、"2 ギター" を追加する。

また、この表に記載されている "略号" にも、いくつか誤りがあった。これについては、この本の 136 ページに掲載したプログラム (READFM.BAS) を実行すると、正しい略号を表示するようになっている。それぞれ実際に音色を演奏させながら、略号を確認していってほしい。

続いて、148 ページの "VOICE COPY" に関する説明の部分。文章の真ん中あたりに、"ソース (パラメーター 1) に指定できる音色番号は 0~63 のうち OPLL VOICE の欄に指定がある音色の番号です" となっているけど、正しくは "指定がない音色の番号です" ということになる。

これと同様に、その少しあとにある "ソースに OPLL VOICE 欄に指定のない音色の番号を指定すると "Illegal function call" となります" という記述も逆。正しくは "指定のある音色の番号を指定すると……" となるわけだ。

また、151 ページから 158 ページにかけて掲載されていた、OPLL のレジスターの表にも、いくつかの誤りがあった。それらを正したものを図 5.6 にまとめて掲載しておいたので、参考にしてほしい。これをもとに、手元にある MSX2+パワフル活用法の内容を修正しておくと、便利だと思う。

レジスターの説明に関連して、155 ページにある、目的の周波数から F–Number と BLOCK を求める式にも、誤りがあった。一番下に掲載されている、

$$\text{F–Number} = (440 \times 2^{18} \div 50000) \div 2^4 - 1 = 288$$

という式は、正しくは、

$$\text{F–Number} = (440 \times 2^{18} \div 50000) \div 2^{(4-1)} = 288$$

ということになる。

最後に、これは MSX2+ パワフル活用法に限ったことではないのだけど、FM 音源に関する楽器音データの指定方法に、一般に間違った説明がされているようなので、訂正しておく。

音符ごとの音階を指定するのに、00H～5FH までが、それぞれオクターブ 1 の C～オクターブ 7 の B までに対応している、と一般にはいわれているけど、これは間違

図 5.6: OPLL のレジスター一覧

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
00H	AM(M)	VIB(M)	EGT(M)	KSR(M)	Multiple(M)			
01H	AM(C)	VIB(C)	EGT(C)	KSR(C)	Multiple(C)			
02H	KSL(M)		Total Level Modelater					
03H	KSL(C)		空き	DC	DM	Feed Back		
04H	Attack(M)				Decay(M)			
05H	Attack(C)				Decay(C)			
06H	Sustain(M)				Release(M)			
07H	Sustain(C)				Release(C)			
0EH	空き		R	BD	SD	TOM	T–CT	HH
0FH	検査用レジスター							
10H ⋮ 18H	F–number							
20H ⋮ 28H	空き		Sus.	Key	Block			F–number
30H ⋮ 38H	Inst.				Vol.			

リズムモードの場合

	b_7 b_6 b_5 b_4	b_3 b_2 b_1 b_0
36H	空き	Bass Drum
37H	Hi Hat	Snare Drum
38H	Tom Tom	Top Cymbal

表中で (M) となっている部分は、モジュレーターとして働くオペレーター 0 を、(C) となっているものは、キャリアーとして働くオペレーター 1 を示している。詳細については、"MSX2+パワフル活用法" または、ヤマハの技術資料を参照のこと。

い。正しくは00Hは休符となり、続く01H~5FHまでが、オクターブ1のC~オクターブ7のA#に対応している。

5.4.2 MSX–MUSICの音色データ一覧

プログラマーのみなさまのご要望にお答えして、MSX–MUSICのROMに内蔵された音色データの、ダンプリストを掲載しよう。

表5.9の左側に掲載したのは、BASICの"CALL VOICE COPY"ステートメントで得られる32バイトの音色データから、OPLLのレジスター0~7に書き込まれる8バイトのデータを抜き出したもの。音色データには、"ボイス移調"と呼ばれる音の高さを制御する2バイトのデータも含まれているけど、OPLLレジスターに直接書き込まれるデータではないので、掲載を省略した。

音色60と音色61は、この表ではまったく同じにみえるけど、ボイス移調の値が異なるので、実際には違う音色になっている。また、表中で"using data of OPLL"と書かれている音色番号については、OPLLに内蔵された音色が使われるので、ROMには音色データが含まれていない。

さて、FM–BIOSで得られる63種類のROM内蔵音色データは、このBASICのROM内蔵音色データと共通だと、みんなが信じて疑わなかった。ところが、実際には異なっているという事実がいまになって判明した。というわけで、表5.9の右側は、FM–BIOSの"RDDATA"機能を使って得られた、各音色につき8バイトの音色データだ。

表5.9の左右を比べるとわかるように、拡張BASICとFM–BIOSについて音色番号と名称の対応は共通だけど、音色データは微妙に違っている。そのため、BASICのMMLを使って曲を試作して、そのデータをFM–BIOS用に変換するような場合に、音色の違いが問題になるかもしれない。

なお、FM–BIOSの大部分の音色データについて、レジスター3に書き込まれる値が20Hとなっていることを、不思議に感じる人もいるかもしれない。でも、レジスター3のビット5は"空き"になっているので、レジスター3に書き込まれる値が20Hであっても、そうでなくても、実際に演奏される音色は同じになる。

表 5.9: 音色データ一覧

番号	音色名	拡張 BASIC の音色データ		FM-BIOS の音色データ	
0	Piano 1	using data of OPLL(3)		31 11 0E 20	D9 B2 11 F4
1	Piano 2	30 10 0F 04	D9 B2 10 F4	30 10 0F 20	D9 B2 10 F3
2	Violin	using data of OPLL(1)		61 61 12 20	B4 56 14 17
3	Flute 1	using data of OPLL(4)		61 31 20 20	6C 43 18 26
4	Clarinet	using data of OPLL(5)		A2 30 A0 20	88 54 14 06
5	Oboe	using data of OPLL(6)		31 34 20 20	72 56 0A 1C
6	Trumpet	using data of OPLL(7)		31 71 16 20	51 52 26 24
7	Pipe Organ 1	34 30 37 06	50 30 76 06	34 30 37 20	50 30 76 06
8	Xylophone	17 52 18 05	88 D9 66 24	17 52 18 20	88 D9 66 24
9	Organ	using data of OPLL(8)		E1 63 0A 20	FC F8 28 29
10	Guitar	using data of OPLL(2)		02 41 15 20	A3 A3 75 05
11	Santool 1	19 53 0C 06	C7 F5 11 03	19 53 0C 20	C7 F5 11 03
12	Electric Piano 1	using data of OPLL(15)		23 43 09 20	DD BF 4A 05
13	Clavicode 1	03 09 11 06	D2 B4 F5 F6	03 09 11 20	D2 B4 F4 F5
14	Harpsicode 1	using data of OPLL(11)		01 00 06 20	A3 E2 F4 F4
15	Harpsicode 2	01 01 11 06	C0 B4 01 F7	01 01 11 20	C0 B4 01 F6
16	Vibraphone	using data of OPLL(12)		F9 F1 24 20	95 D1 E5 F2
17	Koto 1	13 11 0C 06	FC D2 33 84	13 11 0C 20	FC D2 33 83
18	Taiko	01 10 0E 07	CA E6 44 24	01 10 0E 20	CA E6 44 24
19	Engine 1	E0 F4 1B 87	11 F0 04 08	E0 F4 1B 20	11 F0 04 08
20	UFO	FF 70 19 07	50 1F 05 01	FF 70 19 20	50 1F 05 01
21	Synthesizer Bell	13 11 11 07	FA F2 21 F5	13 11 11 20	FA F2 21 F4
22	Chime	A6 42 10 05	FB B9 11 02	A6 42 10 20	FB B9 11 02
23	Synthesizer Bass	using data of OPLL(13)		40 31 89 20	C7 F9 14 04
24	Synthesizer	using data of OPLL(10)		42 44 0B 20	94 B0 33 F6
25	Synthesizer Percussion	01 03 0B 07	BA D9 25 06	01 03 0B 20	BA D9 25 06
26	Synthesizer Rhythm	40 00 00 07	FA D9 37 04	40 00 00 20	FA D9 37 04
27	Harm Drum	02 03 09 07	CB FF 39 06	02 03 09 20	CB FF 39 06
28	Cowbell	18 11 09 05	F8 F5 26 26	18 11 09 20	F8 F5 26 26
29	Close Hi-hat	0B 04 09 07	F0 F5 01 27	0B 04 09 20	F0 F5 01 27
30	Snare Drum	40 40 07 07	D0 D6 01 27	40 40 07 20	D0 D6 01 27
31	Bass Drum	00 01 07 06	CB E3 36 25	00 01 07 20	CB E3 36 25

番号	音色名	拡張 BASIC の音色データ		FM–BIOS の音色データ	
32	Piano 3	11 11 08 04	FA B2 20 F5	11 11 08 20	FA B2 20 F4
33	Electric Piano 2	using data of OPLL(14)		11 11 11 20	C0 B2 01 F4
34	Santool 2	19 53 15 07	E7 95 21 03	19 53 15 20	E7 95 21 03
35	Brass	30 70 19 07	42 62 26 24	30 70 19 20	42 62 26 24
36	Flute 2	62 71 25 07	64 43 12 26	62 71 25 20	64 43 12 26
37	Clavicode 2	21 03 0B 05	90 D4 02 F6	21 03 0B 20	90 D4 02 F5
38	Clavicode 3	01 03 0A 05	90 A4 03 F6	01 03 0A 20	90 A4 03 F5
39	Koto 2	43 53 0E 85	B5 E9 85 04	43 53 0E 20	B5 E9 84 04
40	Pipe Organ 2	34 30 26 06	50 30 76 06	34 30 26 20	50 30 76 06
41	PohdsPLA	73 33 5A 06	99 F5 14 15	73 33 5A 20	99 F5 14 15
42	RohdsPRA	73 13 16 05	F9 F5 33 03	73 13 16 20	F9 F5 33 03
43	Orch L	61 21 15 07	76 54 23 06	61 21 15 20	76 54 23 06
44	Orch R	63 70 1B 07	75 4B 45 15	63 70 1B 20	75 4B 45 15
45	Synthesizer Violin	61 A1 0A 05	76 54 12 07	61 A1 0A 20	76 54 12 07
46	Synthesizer Organ	61 78 0D 05	85 F2 14 03	61 78 0D 20	85 F2 14 03
47	Synthesizer Brass	31 71 15 07	B6 F9 03 26	31 71 15 20	B6 F9 03 26
48	Tube	using data of OPLL(9)		61 71 0D 20	75 F2 18 03
49	Shamisen	03 0C 14 06	A7 FC 13 15	03 0C 14 20	A7 FC 13 15
50	Magical	13 32 81 03	20 85 03 B0	13 32 80 20	20 85 03 AF
51	Huwawa	F1 31 17 05	23 40 14 09	F1 31 17 20	23 40 14 09
52	Wander Flat	F0 74 17 47	5A 43 06 FD	F0 74 17 20	5A 43 06 FC
53	Hardrock	20 71 0D 06	C1 D5 56 06	20 71 0D 20	C1 D5 56 06
54	Machine	30 32 06 06	40 40 04 74	30 32 06 20	40 40 04 74
55	Machine V	30 32 03 03	40 40 04 74	30 32 03 20	40 40 04 74
56	Comic	01 08 0D 07	78 F8 7F FA	01 08 0D 20	78 F8 7F F9
57	SE–Comic	C8 C0 0B 05	76 F7 11 FA	C8 C0 0B 20	76 F7 11 F9
58	SE–Laser	49 40 0B 07	B4 F9 00 05	49 40 0B 20	B4 F9 FF 05
59	SE–Noise	CD 42 0C 06	A2 F0 00 01	CD 42 0C 20	A2 F0 00 01
60	SE–Star 1	51 42 13 07	13 10 42 01	51 42 13 20	13 10 42 01
61	SE–Star 2	51 42 13 07	13 10 42 01	51 42 13 20	13 10 42 01
62	Engine 2	30 34 12 06	23 70 26 02	30 34 12 20	23 70 26 02
63	Silence	00 00 00 00	00 00 00 00	00 00 FF 20	00 00 FF FF

APPENDIX

R800 インストラクション表

1991 年 1 月 24 日

株式会社アスキー
システム事業部、MSX マガジン編集部

マシン語レベルのプログラミングに燃える人なら、ぜひとも挑戦してほしいのが、R800 でのプログラム開発。ニーモニックや命令動作、マシン語コードを記した、インストラクション表を掲載したので活用してほしい。さあ、R800 の速度を活かしたプログラムはできるかな？

A.1 インストラクション表はこうして使おう

この表は、命令の種類ごとに分類して、R800のインストラクションをまとめたもの。表中の“ニーモニック”は各命令の名前を現わし、“命令動作”でその動作内容を簡潔に示している。

命令動作の欄で“←”とあるのは、右側の内容を左側に代入することを、カッコでくくられたものは、くくられたレジスターなどで示されるメモリーの内容を、それぞれ意味している。たとえば、

```
r ← [.hl]
```

とあるのは、.hl レジスターで示されるアドレスのメモリーの内容を、8 ビットレジスターに代入するということだ。ただし入出力命令の [n] と [.c] は、対応する入出力ポートの番号を意味している。

“フラグ”の欄は各フラグの動作を、“オペコード”はそれぞれの命令に対するマシン語コードを、2 進数と 16 進数で記したもの。その右側の“B”と“C”は、各命令の長さ (バイト数) と、命令を実行するのに要するクロック数を、それぞれ現わしている。

このほか、インストラクション表に出てくる略号に関して、次の凡例にまとめておいたので参考にしてほしい。また、表に記載されたニーモニックが Z80 と違っている理由は、それがザイログ社の著作物だから。といっても、R800 で追加された乗算命令や、Z80 で正式に動作が保証されていなかった命令以外は、ニーモニックの違いがあるにせよ、命令動作はすべて同じになっている。Z80 のインストラクション表と見比べながら、プログラムしていってほしい。

.a{7}	レジスター.aの最上位ビット
.a{4..7}	レジスター.aのビット4-7
;	動作の区切り
.de:.hl	上位16ビットが.de、下位16ビットが.hlに入る、32ビット整数
[.ix+d]	.ixに8ビットの符号つき変位を足した値が示すアドレス
C	キャリーフラグ
Z	ゼロフラグ
P/V	パリティー・オーバーフローフラグ
S	サインフラグ
N	減算フラグ
H	ハーフキャリーフラグ
•	フラグは変化しない
↕	フラグは実行結果により変化する
0	フラグは0
1	フラグは1
?	不定になる
V	オーバーフローフラグとして使われる
P	パリティーフラグとして使われる
IFF	割り込みフリップフロップの値が入る
r,r'	8ビットレジスター、.a,.b,.c,.d,.e,.h,.l
u,u'	8ビットレジスター、.a,.b,.c,.d,.e,.ixh,.ixl
v,v'	8ビットレジスター、.a,.b,.c,.d,.e,.iyh,.iyl
p	8ビットレジスター、.ixh,.ixl
q	8ビットレジスター、.iyh,.iyl
ss	16ビットレジスター、.bc,.de,.hl,.sp
pp	16ビットレジスター、.bc,.de,.ix,.sp
rr	16ビットレジスター、.bc,.de,.iy,.sp
qq	16ビットレジスター、.bc,.de,.hl,.af
e	short br系の命令の飛び先アドレスへの差分、8ビットの符号つき即値(+127〜−128)
k	brk命令の飛び先アドレス、00h,08h,10h,18h,20h,28h,30h,38h
nn	16ビットの即値、もしくは絶対アドレス
n	8ビットの即値
b	ビット演算命令の第何ビットかを示す値
NOT	ビットを反転する
∨	ビットのORをとる
⊻	ビットのXORをとる
∧	ビットのANDをとる
tmp	一時的に値を待避する
B	命令のバイト数
C	命令の実行に必要な最小クロック数

分岐命令、コール命令でクロック数がふたつ書いてあるものは、上が条件が成立しないとき、下が条件が成立したときを意味する。

また、入出力命令でクロック数がふたつ書いてあるものは、上がまだ転送が終わらないとき、下が転送が終わったときをそれぞれ意味している。

ここに記す命令表のクロック数は、SYSCLK換算でXTALの発振周波数の4分の1。またノーウェイトで実行したときの値で、DRAM上で実行したときはページブレークやリフレッシュにより、自動的にウェイトが挿入される。

A.2 8ビット移動命令

ニーモニック	命令動作	flags	オペコード		B	C
		S Z H P/V N C	76543210	Hex		
ld r,r'	r←r'	• • • • • •	01 r r'		1	1
ld r,n	r←n	• • • • • •	00 r 110 ← n →		2	2
ld r,[.hl]	r←[.hl]	• • • • • •	01 r 110		1	2
ld r,[.ix+d]	r←[.ix+d]	• • • • • •	11011101 01 r 110 ← d →	DD	3	5
ld r,[.iy+d]	r←[.iy+d]	• • • • • •	11111101 01 r 110 ← d →	FD	3	5
ld [.hl],r	[.hl]←r	• • • • • •	01110 r		1	2
ld [.ix+d],r	[.ix+d]←r	• • • • • •	11011101 01110 r ← d →	DD	3	5
ld [.iy+d],r	[.iy+d]←r	• • • • • •	11111101 01110 r ← d →	FD	3	5
ld u,u'	u←u'	• • • • • •	11011101 01 u u'	DD	2	2
ld v,v'	v←v'	• • • • • •	11111101 01 v v'	FD	2	2
ld u,n	u←n	• • • • • •	11011101 00 u 110 ← n →	DD	3	3
ld v,n	v←n	• • • • • •	11111101 00 v 110 ← n →	FD	3	3
ld [.hl],n	[.hl]←n	• • • • • •	00110110 ← n →	36	2	3
ld [.ix+d],n	[.ix+d]←n	• • • • • •	11011101 00110110 ← d → ← n →	DD 36	4	5
ld [.iy+d],n	[.iy+d]←n	• • • • • •	11111101 00110110 ← d → ← n →	FD 36	4	5

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
ld .a,.i	.a←.i	↕ ↕ 0 IFF 0 •	11101101 01010111	ED 57	2	2
ld .a,.r	.a←.r	↕ ↕ 0 IFF 0 •	11101101 01011111	ED 5F	2	2
ld .i,.a	.i←.a	• • • • • •	11101101 01000111	ED 47	2	2
ld .r,.a	.r←.a	• • • • • •	11101101 01001111	ED 4F	2	2
ld .a,[.bc]	.a←[.bc]	• • • • • •	00001010	0A	1	2
ld .a,[.de]	.a←[.de]	• • • • • •	00011010	1A	1	2
ld .a,[nn]	.a←[nn]	• • • • • •	00111010 ← nn_l → ← nn_h →	3A	3	4
ld [.bc],.a	[.bc]←.a	• • • • • •	00000010	02	1	2
ld [.de],.a	[.de]←.a	• • • • • •	00010010	12	1	2
ld [nn],.a	[nn]←.a	• • • • • •	00110010 ← nn_l → ← nn_h →	32	3	4

	000	001	010	011	100	101	110	111
r	.b	.c	.d	.e	.h	.l		.a
u	.b	.c	.d	.e	.ixh	.ixl		.a
v	.b	.c	.d	.e	.iyh	.iyl		.a

A.3 16 ビット移動命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
ld ss,nn	ss←nn	• • • • • •	00 ss 0001 ← nn_l → ← nn_h →		3	3
ld .ix,nn	.ix←nn	• • • • • •	11011101 00100001 ← nn_l → ← nn_h →	DD 21	4	4
ld .iy,nn	.iy←nn	• • • • • •	11111101 00100001 ← nn_l → ← nn_h →	FD 21	4	4
ld .sp,.hl	.sp←.hl	• • • • • •	11111001	F9	1	1
ld .sp,.ix	.sp←.ix	• • • • • •	11011101 11111001	DD F9	2	2
ld .sp,.iy	.sp←.iy	• • • • • •	11111101 11111001	FD F9	2	2

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
ld ss,[nn]	ss_h←[nn+1] ss_l←[nn]	• • • • • •	11 101 101 01 ss 1 011 ← nn_l → ← nn_h →	ED	4	6
ld .hl,[nn]	.h←[nn+1] .l←[nn]	• • • • • •	00 101 010 ← nn_l → ← nn_h →	2A	3	5
ld .ix,[nn]	.ixh←[nn+1] .ixl←[nn]	• • • • • •	11 011 101 00 101 010 ← nn_l → ← nn_h →	DD 2A	4	6
ld .iy,[nn]	.iyh←[nn+1] .iyl←[nn]	• • • • • •	11 111 101 00 101 010 ← nn_l → ← nn_h →	FD 2A	4	6
ld [nn],ss	[nn+1]←ss_h [nn]←ss_l	• • • • • •	11 101 101 01 ss 0 011 ← nn_l → ← nn_h →	ED	4	6
ld [nn],.hl	[nn+1]←.h [nn]←.l	• • • • • •	00 100 010 ← nn_l → ← nn_h →	22	3	5
ld [nn],.ix	[nn+1]←.ixh [nn]←.ixl	• • • • • •	11 011 101 00 100 010 ← nn_l → ← nn_h →	DD 22	4	6
ld [nn],.iy	[nn+1]←.iyh [nn]←.iyl	• • • • • •	11 111 101 00 100 010 ← nn_l → ← nn_h →	FD 22	4	6

	00	01	10	11
ss	.bc	.de	.hl	.sp

A.4 交換命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76 543 210	Hex	B	C
xch .de,.hl	.de↔.hl	• • • • • •	11 101 011	EB	1	1
xch .af,.af'	.af↔.af'	↕ ↕ ↕ ↕ ↕ ↕	00 001 000	08	1	1
xch [.sp],.hl	.l↔[.sp];.h↔[.sp+1]	• • • • • •	11 100 011	E3	1	5
xch [.sp],.ix	.ixl↔[.sp] .ixh↔[.sp+1]	• • • • • •	11 011 101 11 100 011	DD E3	2	6
xch [.sp],.iy	.iyl↔[.sp] .iyh↔[.sp+1]	• • • • • •	11 111 101 11 100 011	FD E3	2	6
xchx	.bc↔.bc';.de↔.de';.hl↔.hl'	• • • • • •	11 011 001	D9	1	1

A.5 スタック操作命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76 543 210	Hex	B	C
push qq	[.sp−2]←qq_l;[.sp−1]←qq_h .sp←.sp−2	• • • • • •	11 qq0 101		1	4
push .ix	[.sp−2]←.ixl;[.sp−1]←.ixh .sp←.sp−2	• • • • • •	11 011 101 11 100 101	DD E5	2	5
push .iy	[.sp−2]←.iyl;[.sp−1]←.iyh .sp←.sp−2	• • • • • •	11 111 101 11 100 101	FD E5	2	5
pop qq	qq_l←[.sp];qq_h←[.sp+1] .sp←.sp+2	• • • • • •	11 qq0 001		1	3
pop .ix	.ixl←[.sp];.ixh←[.sp+1] .sp←.sp+2	• • • • • •	11 011 101 11 100 001	DD E1	2	4
pop .iy	.iyl←[.sp];.iyh←[.sp+1] .sp←.sp+2	• • • • • •	11 111 101 11 100 001	FD E1	2	4

	00	01	10	11
qq	.bc	.de	.hl	.af

pop .af のときは flags はすべて変化する

A.6 ブロック転送命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
move [.hl++],[.de++]	[.de]←[.hl];.de←.de+1 .hl←.hl+1;.bc←.bc−1	• • 0 ↕ 0 • (*1)	11101101 10100000	ED A0	2	4
move [.hl−−],[.de−−]	[.de]←[.hl];.de←.de−1 .hl←.hl−1;.bc←.bc−1	• • 0 ↕ 0 • (*1)	11101101 10101000	ED A8	2	4
movem [.hl++],[.de++]	repeat;[.de]←[.hl];.de←.de+1 .hl←.hl+1;.bc←.bc−1;until .bc=0	• • 0 0 0 •	11101101 10110000	ED B0	2	4
movem [.hl−−],[.de−−]	repeat;[.de]←[.hl];.de←.de−1 .hl←.hl−1;.bc←.bc−1;until .bc=0	• • 0 0 0 •	11101101 10111000	ED B8	2	4

*1.bc−1=0 のとき 0、その他 1

A.7 ブロックサーチ命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
cmp .a,[.hl++]	.a−[.hl];.hl←.hl+1 .bc←.bc−1	↕ ↕ ↕ ↕ 1 • (Z:*2, P/V:*1)	11101101 10100001	ED A1	2	4
cmp .a,[.hl−−]	.a−[.hl];.hl←.hl−1 .bc←.bc−1	↕ ↕ ↕ ↕ 1 • (Z:*2, P/V:*1)	11101101 10101001	ED A9	2	4
cmpm .a,[.hl++]	repeat;.a−[.hl];.hl←.hl+1 .bc←.bc−1;until .bc=0 or .a=[.hl]	↕ ↕ ↕ ↕ 1 • (Z:*2, P/V:*1)	11101101 10110001	ED B1	2	5
cmpm .a,[.hl−−]	repeat;.a−[.hl];.hl←.hl−1 .bc←.bc−1;until .bc=0 or .a=[.hl]	↕ ↕ ↕ ↕ 1 • (Z:*2, P/V:*1)	11101101 10111001	ED B9	2	5

*1.bc−1=0 のとき 0、その他 1

*2.a=[.hl] のとき 1、その他 0

A.8 乗算命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
mulub .a,r	.hl←.a*r	0 ↕ • 0 • ↕	11101101 11 r 001	ED	2	14
muluw .hl,ss	.de:.hl←.hl*ss	0 ↕ • 0 • ↕	11101101 11 ss 0011	ED	2	36

mulub では r が.b,.c,.d,.e のとき以外は動作が保証されない

muluw では ss が.bc,.sp のとき以外は動作が保証されない

A.9 加算命令

ニーモニック	命令動作	flags	オペコード		B	C
		S Z H P/V N C	76543210	Hex		
add .a,r	.a←.a+r	↕ ↕ ↕ V 0 ↕	10000 r		1	1
add .a,p	.a←.a+p	↕ ↕ ↕ V 0 ↕	11011101 10000 p	DD	2	2
add .a,q	.a←.a+q	↕ ↕ ↕ V 0 ↕	11111101 10000 q	FD	2	2
add .a,[.hl]	.a←.a+[.hl]	↕ ↕ ↕ V 0 ↕	10000110	86	1	2
add .a,[.ix+d]	.a←.a+[.ix+d]	↕ ↕ ↕ V 0 ↕	11011101 10000110 ← d →	DD 86	3	5
add .a,[.iy+d]	.a←.a+[.iy+d]	↕ ↕ ↕ V 0 ↕	11111101 10000110 ← d →	FD 86	3	5
add .a,n	.a←.a+n	↕ ↕ ↕ V 0 ↕	11000110 ← n →	C6	2	2
addc.a,r	.a←.a+r+c	↕ ↕ ↕ V 0 ↕	10001 r		1	1
addc.a,p	.a←.a+p+c	↕ ↕ ↕ V 0 ↕	11011101 10001 p	DD	2	2
addc.a,q	.a←.a+q+c	↕ ↕ ↕ V 0 ↕	11111101 10001 q	FD	2	2
addc.a,[.hl]	.a←.a+[.hl]+c	↕ ↕ ↕ V 0 ↕	10001110	8E	1	2
addc.a,[.ix+d]	.a←.a+[.ix+d]+c	↕ ↕ ↕ V 0 ↕	11011101 10001110 ← d →	DD 8E	3	5
addc.a,[.iy+d]	.a←.a+[.iy+d]+c	↕ ↕ ↕ V 0 ↕	11111101 10001110 ← d →	FD 8E	3	5
addc.a,n	.a←.a+n+c	↕ ↕ ↕ V 0 ↕	11001110 ← n →	CE	2	2
addc.hl,ss	.hl←.hl+ss+c	↕ ↕ ? V 0 ↕	11101101 01 ss 1010	ED	2	2
add .hl,ss	.hl←.hl+ss	• • ? • 0 ↕	00 ss 1001		1	1
add .ix,pp	.ix←.ix+pp	• • ? • 0 ↕	11011101 00 pp 1001	DD	2	2
add .iy,rr	.iy←.iy+rr	• • ? • 0 ↕	11111101 00 rr 1001	FD	2	2

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76 543 210	 Hex	B	C
inc r	r←r+1	↕ ↕ ↕ V 0 •	00 r 100		1	1
inc p	p←p+1	↕ ↕ ↕ V 0 •	11 011 101 00 p 100	DD	2	2
inc q	q←q+1	↕ ↕ ↕ V 0 •	11 111 101 00 q 100	FD	2	2
inc [.hl]	[.hl]←[.hl]+1	↕ ↕ ↕ V 0 •	00 110 100	34	1	4
inc [.ix+d]	[.ix+d]←[.ix+d]+1	↕ ↕ ↕ V 0 •	11 011 101 00 110 100 ← d →	DD 34	3	7
inc [.iy+d]	[.iy+d]←[.iy+d]+1	↕ ↕ ↕ V 0 •	11 111 101 00 110 100 ← d →	FD 34	3	7
inc ss	ss←ss+1	• • • • • •	00 ss 0 011		1	1
inc .ix	.ix←.ix+1	• • • • • •	11 011 101 00 100 011	DD 23	2	2
inc .iy	.iy←.iy+1	• • • • • •	11 111 101 00 100 011	FD 23	2	2

	00	01	10	11
ss	.bc	.de	.hl	.sp
pp	.bc	.de	.ix	.sp
rr	.bc	.de	.iy	.sp

	000	001	010	011	100	101	110	111
p					.ixh	.ixl		
q					.iyh	.iyl		

A.10　減算命令

ニーモニック	命令動作	flags	オペコード		B	C
		S Z H P/V N C	76 543 210	Hex		
sub .a,r	.a←.a−r	↕ ↕ ↕ V 1 ↕	10010 r		1	1
sub .a,p	.a←.a−p	↕ ↕ ↕ V 1 ↕	11 011 101 10010 p	DD	2	2
sub .a,q	.a←.a−q	↕ ↕ ↕ V 1 ↕	11 111 101 10010 q	FD	2	2
sub .a,[.hl]	.a←.a−[.hl]	↕ ↕ ↕ V 1 ↕	10 010 110	96	1	2
sub .a,[.ix+d]	.a←.a−[.ix+d]	↕ ↕ ↕ V 1 ↕	11 011 101 10 010 110 ← d →	DD 96	3	5
sub .a,[.iy+d]	.a←.a−[.iy+d]	↕ ↕ ↕ V 1 ↕	11 111 101 10 010 110 ← d →	FD 96	3	5
sub .a,n	.a←.a−n	↕ ↕ ↕ V 1 ↕	11 010 110 ← n →	D6	2	2
subc.a,r	.a←.a−r−c	↕ ↕ ↕ V 1 ↕	10011 r		1	1
subc.a,p	.a←.a−p−c	↕ ↕ ↕ V 1 ↕	11 011 101 10011 p	DD	2	2
subc.a,q	.a←.a−q−c	↕ ↕ ↕ V 1 ↕	11 111 101 10011 q	FD	2	2
subc.a,[.hl]	.a←.a−[.hl]−c	↕ ↕ ↕ V 1 ↕	10 011 110	9E	1	2
subc.a,[.ix+d]	.a←.a−[.ix+d]−c	↕ ↕ ↕ V 1 ↕	11 011 101 10 011 110 ← d →	DD 9E	3	5
subc.a,[.iy+d]	.a←.a−[.iy+d]−c	↕ ↕ ↕ V 1 ↕	11 111 101 10 011 110 ← d →	FD 9E	3	5
subc.a,n	.a←.a−n−c	↕ ↕ ↕ V 1 ↕	11 011 110 ← n →	DE	2	2
subc.hl,ss	.hl←.hl−ss−c	↕ ↕ ? V 1 ↕	11 101 101 01 ss 0010	ED	2	2

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
dec r	r←r−1	↕ ↕ ↕ V 1 •	00 r 101		1	1
dec p	p←p−1	↕ ↕ ↕ V 1 •	11011101 00 p 101	DD	2	2
dec q	q←q−1	↕ ↕ ↕ V 1 •	11111101 00 q 101	FD	2	2
dec [.hl]	[.hl]←[.hl]−1	↕ ↕ ↕ V 1 •	00110101	35	1	4
dec [.ix+d]	[.ix+d]←[.ix+d]−1	↕ ↕ ↕ V 1 •	11011101 00110101 ← d →	DD 35	3	7
dec [.iy+d]	[.iy+d]←[.iy+d]−1	↕ ↕ ↕ V 1 •	11111101 00110101 ← d →	FD 35	3	7
dec ss	ss←ss−1	• • • • • •	00 ss 1011		1	1
dec .ix	.ix←.ix−1	• • • • • •	11011101 00101011	DD 2B	2	2
dec .iy	.iy←.iy−1	• • • • • •	11111101 00101011	FD 2B	2	2

A.11 比較命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
cmp .a,r	.a−r	↕ ↕ ↕ V 1 ↕	10111 r		1	1
cmp .a,p	.a−p	↕ ↕ ↕ V 1 ↕	11011101 10111 p	DD	2	2
cmp .a,q	.a−q	↕ ↕ ↕ V 1 ↕	11111101 10111 q	FD	2	2
cmp .a,[.hl]	.a−[.hl]	↕ ↕ ↕ V 1 ↕	10111110	BE	1	2
cmp .a,[.ix+d]	.a−[.ix+d]	↕ ↕ ↕ V 1 ↕	11011101 10111110 ← d →	DD BE	3	5
cmp .a,[.iy+d]	.a−[.iy+d]	↕ ↕ ↕ V 1 ↕	11111101 10111110 ← d →	FD BE	3	5
cmp .a,n	.a−n	↕ ↕ ↕ V 1 ↕	11111110 ← n →	FE	2	2

A.12 論理演算命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
and .a,r	.a←.a∧r	↕ ↕ 1 P 0 0	10100 r		1	1
and .a,p	.a←.a∧p	↕ ↕ 1 P 0 0	11011101 10100 p	DD	2	2
and .a,q	.a←.a∧q	↕ ↕ 1 P 0 0	11111101 10100 q	FD	2	2
and .a,[.hl]	.a←.a∧[.hl]	↕ ↕ 1 P 0 0	10100110	A6	1	2
and .a,[.ix+d]	.a←.a∧[.ix+d]	↕ ↕ 1 P 0 0	11011101 10100110 ← d →	DD A6	3	5
and .a,[.iy+d]	.a←.a∧[.iy+d]	↕ ↕ 1 P 0 0	11111101 10100110 ← d →	FD A6	3	5
and .a,n	.a←.a∧n	↕ ↕ 1 P 0 0	11100110 ← n →	E6	2	2
or .a,r	.a←.a∨r	↕ ↕ 0 P 0 0	10110 r		1	1
or .a,p	.a←.a∨p	↕ ↕ 0 P 0 0	11011101 10110 p	DD	2	2
or .a,q	.a←.a∨q	↕ ↕ 0 P 0 0	11111101 10110 q	FD	2	2
or .a,[.hl]	.a←.a∨[.hl]	↕ ↕ 0 P 0 0	10110110	B6	1	2
or .a,[.ix+d]	.a←.a∨[.ix+d]	↕ ↕ 0 P 0 0	11011101 10110110 ← d →	DD B6	3	5
or .a,[.iy+d]	.a←.a∨[.iy+d]	↕ ↕ 0 P 0 0	11111101 10110110 ← d →	FD B6	3	5
or .a,n	.a←.a∨n	↕ ↕ 0 P 0 0	11110110 ← n →	F6	2	2
xor .a,r	.a←.a∀r	↕ ↕ 0 P 0 0	10101 r		1	1
xor .a,p	.a←.a∀p	↕ ↕ 0 P 0 0	11011101 10101 p	DD	2	2
xor .a,q	.a←.a∀q	↕ ↕ 0 P 0 0	11111101 10101 q	FD	2	2
xor .a,[.hl]	.a←.a∀[.hl]	↕ ↕ 0 P 0 0	10101110	AE	1	2
xor .a,[.ix+d]	.a←.a∀[.ix+d]	↕ ↕ 0 P 0 0	11011101 10101110 ← d →	DD AE	3	5
xor .a,[.iy+d]	.a←.a∀[.iy+d]	↕ ↕ 0 P 0 0	11111101 10101110 ← d →	FD AE	3	5
xor .a,n	.a←.a∀n	↕ ↕ 0 P 0 0	11101110 ← n →	EE	2	2

A.13 ビット操作命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	 Hex	B	C
bit b,r	z←NOT $r_{\{b\}}$	? ↕ 1 ? 0 •	11001011 01 b r	CB	2	2
bit b,[.hl]	z←NOT $[.hl]_{\{b\}}$	? ↕ 1 ? 0 •	11001011 01 b 110	CB	2	3
bit b,[.ix+d]	z←NOT $[.ix+d]_{\{b\}}$	? ↕ 1 ? 0 •	11011101 11001011 ← d → 01 b 110	DD CB	4	5
bit b,[.iy+d]	z←NOT $[.iy+d]_{\{b\}}$	? ↕ 1 ? 0 •	11111101 11001011 ← d → 01 b 110	FD CB	4	5
set b,r	$r_{\{b\}}$←1	• • • • • •	11001011 11 b r	CB	2	2
set b,[.hl]	$[.hl]_{\{b\}}$←1	• • • • • •	11001011 11 b 110	CB	2	5
set b,[.ix+d]	$[.ix+d]_{\{b\}}$←1	• • • • • •	11011101 11001011 ← d → 11 b 110	DD CB	4	7
set b,[.iy+d]	$[.iy+d]_{\{b\}}$←1	• • • • • •	11111101 11001011 ← d → 11 b 110	FD CB	4	7
clr b,r	$r_{\{b\}}$←0	• • • • • •	11001011 10 b r	CB	2	2
clr b,[.hl]	$[.hl]_{\{b\}}$←0	• • • • • •	11001011 10 b 110	CB	2	5
clr b,[.ix+d]	$[.ix+d]_{\{b\}}$←0	• • • • • •	11011101 11001011 ← d → 10 b 110	DD CB	4	7
clr b,[.iy+d]	$[.iy+d]_{\{b\}}$←0	• • • • • •	11111101 11001011 ← d → 10 b 110	FD CB	4	7

A.14 ローテイト命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
rola	C←.a{7};.a←.a*2;.a{0}←C	• • 0 • 0 ↕	00000111	07	1	1
rora	C←.a{0};.a←.a/2;.a{7}←C	• • 0 • 0 ↕	00001111	0F	1	1
rolca	tmp←C;C←.a{7};.a←.a*2;.a{0}←tmp	• • 0 • 0 ↕	00010111	17	1	1
rorca	tmp←C;C←.a{0};.a←.a/2;.a{7}←tmp	• • 0 • 0 ↕	00011111	1F	1	1
rol r	C←r{7} r←r*2;r{0}←C	↕ ↕ 0 P 0 ↕	11001011 00000 r	CB	2	2
rol [.hl]	C←[.hl]{7} [.hl]←[.hl]*2;[.hl]{0}←C	↕ ↕ 0 P 0 ↕	11001011 00000110	CB 06	2	5
rol [.ix+d]	C←[.ix+d]{7} [.ix+d]←[.ix+d]*2 [.ix+d]{0}←C	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00000110	DD CB 06	4	7
rol [.iy+d]	C←[.iy+d]{7} [.iy+d]←[.iy+d]*2 [.iy+d]{0}←C	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00000110	FD CB 06	4	7
ror r	C←r{0} r←r/2;r{7}←C	↕ ↕ 0 P 0 ↕	11001011 00001 r	CB	2	2
ror [.hl]	C←[.hl]{0} [.hl]←[.hl]/2;[.hl]{7}←C	↕ ↕ 0 P 0 ↕	11001011 00001110	CB 0E	2	5
ror [.ix+d]	C←[.ix+d]{0} [.ix+d]←[.ix+d]/2 [.ix+d]{7}←C	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00001110	DD CB 0E	4	7
ror [.iy+d]	C←[.iy+d]{0} [.iy+d]←[.iy+d]/2 [.iy+d]{7}←C	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00001110	FD CB 0E	4	7

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76 543 210	Hex	B	C
rolc r	tmp←C;C←r$_{\{7\}}$ r←r*2;r$_{\{0\}}$←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 010 r	CB	2	2
rolc [.hl]	tmp←C;C←[.hl]$_{\{7\}}$ [.hl]←[.hl]*2;[.hl]$_{\{0\}}$←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 010 110	CB 16	2	5
rolc [.ix+d]	tmp←C C←[.ix+d]$_{\{7\}}$ [.ix+d]←[.ix+d]*2 [.ix+d]$_{\{0\}}$←tmp	↕ ↕ 0 P 0 ↕	11 011 101 11 001 011 ← d → 00 010 110	DD CB 16	4	7
rolc [.iy+d]	tmp←C C←[.iy+d]$_{\{7\}}$ [.iy+d]←[.iy+d]*2 [.iy+d]$_{\{0\}}$←tmp	↕ ↕ 0 P 0 ↕	11 111 101 11 001 011 ← d → 00 010 110	FD CB 16	4	7
rorc r	tmp←C;C←r$_{\{0\}}$ r←r/2;r$_{\{7\}}$←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 011 r	CB	2	2
rorc [.hl]	tmp←C;C←[.hl]$_{\{0\}}$ [.hl]←[.hl]/2;[.hl]$_{\{7\}}$←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 011 110	CB 1E	2	5
rorc [.ix+d]	tmp←C C←[.ix+d]$_{\{0\}}$ [.ix+d]←[.ix+d]/2 [.ix+d]$_{\{7\}}$←tmp	↕ ↕ 0 P 0 ↕	11 011 101 11 001 011 ← d → 00 011 110	DD CB 1E	4	7
rorc [.iy+d]	tmp←C C←[.iy+d]$_{\{0\}}$ [.iy+d]←[.iy+d]/2 [.iy+d]$_{\{7\}}$←tmp	↕ ↕ 0 P 0 ↕	11 111 101 11 001 011 ← d → 00 011 110	FD CB 1E	4	7
rol4 [.hl]	tmp←.a$_{\{0..3\}}$;.a$_{\{0..3\}}$←[.hl]$_{\{4..7\}}$ [.hl]$_{\{4..7\}}$←[.hl]$_{\{0..3\}}$;[.hl]$_{\{0..3\}}$←tmp	↕ ↕ 0 P 0 •	11 101 101 11 101 111	ED 6F	2	5
ror4 [.hl]	tmp←.a$_{\{0..3\}}$;.a$_{\{0..3\}}$←[.hl]$_{\{0..3\}}$ [.hl]$_{\{0..3\}}$←[.hl]$_{\{4..7\}}$;[.hl]$_{\{4..7\}}$←tmp	↕ ↕ 0 P 0 •	11 101 101 11 100 111	ED 67	2	5

A.15 シフト命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
shl r shla	C←r{7} r←r*2	↕ ↕ 0 P 0 ↕	11001011 00100 r	CB	2	2
shl [.hl] shla	C←[.hl]{7} [.hl]←[.hl]*2	↕ ↕ 0 P 0 ↕	11001011 00100110	CB 26	2	5
shl [.ix+d] shla	C←[.ix+d]{7} [.ix+d]←[.ix+d]*2	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00100110	DD CB 26	4	7
shl [.iy+d] shla	C←[.iy+d]{7} [.iy+d]←[.iy+d]*2	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00100110	FD CB 26	4	7
shr r	C←r{0} r←r/2	↕ ↕ 0 P 0 ↕	11001011 00111 r	CB	2	2
shr [.hl]	C←[.hl]{0} [.hl]←[.hl]/2	↕ ↕ 0 P 0 ↕	11001011 00111110	CB 3E	2	5
shr [.ix+d]	C←[.ix+d]{0} [.ix+d]←[.ix+d]/2	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00111110	DD CB 3E	4	7
shr [.iy+d]	C←[.iy+d]{0} [.iy+d]←[.iy+d]/2	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00111110	FD CB 3E	4	7
shrar	tmp←r{7};C←r{0} r←r/2;r{7}←tmp	↕ ↕ 0 P 0 ↕	11001011 00101 r	CB	2	2
shra[.hl]	tmp←[.hl]{7};C←[.hl]{0} [.hl]←[.hl]/2;[.hl]{7}←tmp	↕ ↕ 0 P 0 ↕	11001011 00101110	CB 2E	2	5
shra [.ix+d]	tmp←[.ix+d]{7} C←[.ix+d]{0} [.ix+d]←[.ix+d]/2 [.ix+d]{7}←tmp	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00101110	DD CB 2E	4	7
shra [.iy+d]	tmp←[.iy+d]{7} C←[.iy+d]{0} [.iy+d]←[.iy+d]/2 [.iy+d]{7}←tmp	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00101110	FD CB 2E	4	7

shl 命令と shla 命令はまったく同じものなのでオペランドは同一

A.16 分岐命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	Hex	B	C
br nn	.pc←nn	• • • • • •	11000011 ← nn_l → ← nn_h →	C3	3	3
bnz nn	if z=0 .pc←nn	• • • • • •	11000010 ← nn_l → ← nn_h →	C2	3	3
bz nn	if z=1 .pc←nn	• • • • • •	11001010 ← nn_l → ← nn_h →	CA	3	3
bnc nn	if c=0 .pc←nn	• • • • • •	11010010 ← nn_l → ← nn_h →	D2	3	3
bc nn	if c=1 .pc←nn	• • • • • •	11011010 ← nn_l → ← nn_h →	DA	3	3
bpo nn	if P/V=0 .pc←nn	• • • • • •	11100010 ← nn_l → ← nn_h →	E2	3	3
bpe nn	if P/V=1 .pc←nn	• • • • • •	11101010 ← nn_l → ← nn_h →	EA	3	3
bp nn	if s=0 .pc←nn	• • • • • •	11110010 ← nn_l → ← nn_h →	F2	3	3
bm nn	if s=1 .pc←nn	• • • • • •	11111010 ← nn_l → ← nn_h →	FA	3	3
br [.hl]	.pc←[.hl]	• • • • • •	11101001	E9	1	1
br [.ix]	.pc←[.ix]	• • • • • •	11011101 11101001	DD E9	2	2
br [.iy]	.pc←[.iy]	• • • • • •	11111101 11101001	FD E9	2	2

ニーモニック	命令動作	flags	オペコード		B	C
		S Z H P/V N C	76543210	Hex		
short br e	.pc←.pc+e	• • • • • •	00011000 ← e−2 →	18	2	3
short bnz e	if z=0 .pc←.pc+e	• • • • • •	00100000 ← e−2 →	20	2	2 3
short bz e	if z=1 .pc←.pc+e	• • • • • •	00101000 ← e−2 →	28	2	2 3
short bnc e	if c=0 .pc←.pc+e	• • • • • •	00110000 ← e−2 →	30	2	2 3
short bc e	if c=1 .pc←.pc+e	• • • • • •	00111000 ← e−2 →	38	2	2 3
dbnz e	.b←.b−1;if .b≠0 .pc←.pc+e	• • • • • •	00010000 ← e−2 →	10	2	2

A.17 コール命令

ニーモニック	命令動作	flags	オペコード		B	C
		S Z H P/V N C	76543210	Hex		
call nn	[.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11001101 ← nn$_l$ → ← nn$_h$ →	CD	3	5
call nz,nn	if z=0 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11000100 ← nn$_l$ → ← nn$_h$ →	C4	3	3 5
call z,nn	if z=1 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11001100 ← nn$_l$ → ← nn$_h$ →	CC	3	3 5
call nc,nn	if c=0 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11010100 ← nn$_l$ → ← nn$_h$ →	D4	3	3 5
call c,nn	if c=1 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11011100 ← nn$_l$ → ← nn$_h$ →	DC	3	3 5
call po,nn	if P/V=0 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11100100 ← nn$_l$ → ← nn$_h$ →	E4	3	3 5
call pe,nn	if P/V=1 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11101100 ← nn$_l$ → ← nn$_h$ →	EC	3	3 5
call p,nn	if s=0 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11110100 ← nn$_l$ → ← nn$_h$ →	F4	3	3 5
call m,nn	if s=1 [.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc←nn	• • • • • •	11111100 ← nn$_l$ → ← nn$_h$ →	FC	3	3 5

ニーモニック	命令動作	flags	オペコード		B	C
		S Z H P/V N C	76 543 210	Hex		
ret	.pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 001 001	C9	1	3
ret nz	if z=0 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 000 000	C0	1	1 3
ret z	if z=1 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 001 000	C8	1	1 3
ret nc	if c=0 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 010 000	D0	1	1 3
ret c	if c=1 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 011 000	D8	1	1 3
ret po	if P/V=0 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 100 000	E0	1	1 3
ret pe	if P/V=1 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 101 000	E8	1	1 3
ret p	if s=0 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 110 000	F0	1	1 3
ret m	if s=1 .pc$_l$←[.sp];.pc$_h$←[.sp+1];.sp←.sp+2	• • • • • •	11 111 000	F8	1	1 3
reti	interrupt return	• • • • • •	11 101 101 01 001 101	ED 4D	2	5
retn	Non Maskable Interrupt return	• • • • • •	11 101 101 01 000 101	ED 45	2	5
brk k	[.sp−2]←.pc$_l$;[.sp−1]←.pc$_h$ .sp←.sp−2;.pc$_l$←k;.pc$_h$←0	• • • • • •	11 k/8 111		1	4

A.18 入出力命令

ニーモニック	命令動作	flags S Z H P/V N C	オペコード 76543210	 Hex	 B	 C
in .a,[n]	.a←[n]	● ● ● ● ● ●	11011011 ← n →	DB	2	3
in r,[.c]	r←[.c]	↕ ↕ 0 P 0 ●	11101101 01 r 000	ED	2	3
in .f,[.c]	[.c]	↕ ↕ 0 P 0 ●	11101101 01110000	ED 70	2	3
in [.hl++],[.c]	[.hl]←[.c];.b←.b−1 .hl←.hl+1	? ↕ ? ? 1 ● *1	11101101 10100010	ED A2	2	4
in [.hl−−],[.c]	[.hl]←[.c];.b←.b−1 .hl←.hl−1	? ↕ ? ? 1 ● *1	11101101 10101010	ED AA	2	4
inm [.hl++],[.c]	repeat;[.hl]←[.c];.b←.b−1 .hl←.hl+1;until .b=0	? 1 ? ? 1 ●	11101101 10110010	ED B2	2	4 3
inm [.hl−−],[.c]	repeat;[.hl]←[.c];.b←.b−1 .hl←.hl−1;until .b=0	? 1 ? ? 1 ●	11101101 10111010	ED BA	2	4 3
out [n],.a	[n]←.a	● ● ● ● ● ●	11010011 ← n →	D3	2	3
out [.c],r	[.c]←r	● ● ● ● ● ●	11101101 01 r 001	ED	2	3
out [.c],[.hl++]	[.c]←[.hl];.b←.b−1 .hl←.hl+1	? ↕ ? ? 1 ● *1	11101101 10100011	ED A3	2	4
out [.c],[.hl−−]	[.c]←[.hl];.b←.b−1 .hl←.hl−1	? ↕ ? ? 1 ● *1	11101101 10101011	ED AB	2	4
outm [.c],[.hl++]	repeat;[.c]←[.hl];.b←.b−1 .hl←.hl+1;until .b=0	? 1 ? ? 1 ●	11101101 10110011	ED B3	2	4 3
outm [.c],[.hl−−]	repeat;[.c]←[.hl];.b←.b−1 .hl←.hl−1;until .b=0	? 1 ? ? 1 ●	11101101 10111011	ED BB	2	4 3

*1.b−1=0 のとき 1、他は 0

in .f,[.c] は.c レジスターが示すポートの内容によってフラグを変えるだけで、その内容はどこにも格納されない

A.19 CPU 制御命令

ニーモニック	命令動作	flags	オペコード		B	C
		S Z H P/V N C	76 543 210	Hex		
adj .a	adjust to decimal	↕ ↕ ↕ P • ↕	00 100 111	27	1	1
not .a	.a←NOT .a	• • 1 • 1 •	00 101 111	2F	1	1
neg .a	.a←NOT .a+1	↕ ↕ ↕ V 1 ↕	11 101 101 01 000 100	ED 44	2	2
notc	c←NOT c	• • ? • 0 ↕	00 111 111	3F	1	1
setc	c←1	• • 0 • 0 1	00 110 111	37	1	1
nop	NO operation	• • • • • •	00 000 000	00	1	1
halt	HALT	• • • • • •	01 110 110	76	1	2
di	IFF←0	• • • • • •	11 110 011	F3	1	2
ei	IFF←1	• • • • • •	11 111 011	FB	1	1
im 0	interrupt mode 0	• • • • • •	11 101 101 01 000 110	ED 46	2	3
im 1	interrupt mode 1	• • • • • •	11 101 101 01 010 110	ED 56	2	3
im 2	interrupt mode 2	• • • • • •	11 101 101 01 011 110	ED 5E	2	3

索引

ア

カ

サ

タ

ナ

ハ

マ

ヤ

ラ

参考文献

[1] アスキー・マイクロソフトFE本部、日本楽器製造株式会社、**“V9938 MSX-VIDEO テクニカルデータブック”**、アスキー、1985年(絶版)

[2] 株式会社アスキー、**“V9958 仕様書”**、非売品、1988年

[3] アスキー・マイクロソフトFE監修、**“MSX2 テクニカル・ハンドブック”**、アスキー、1986年

[4] 杉谷成一、**“MSX2+パワフル活用法”**、アスキー、1989年

[5] 株式会社アスキー、**“MSX-Datapack”**、アスキー、1991年

■著者略歴

石川直太（いしかわなおた）

横浜国立大学卒業後、アスキーに入社。MSX の開発に携わる。その後、東京理科大学理学部第二部数学課、同大学院理学研究課修士課程を卒業。現在は慶應義塾大学大学院理工学研究課で、後期博士課程に在学中。MSX マガジンに連載された歴代のテクニカル記事の筆者でもある。naota@slab. sfc. keio. ac. jp

MSX turbo R テクニカル・ハンドブック

1991年7月31日　初版発行
定価2,500円(本体2,427円)

著　者　石川直太
発行者　塚本慶一郎
編集人　小島文隆
発行所　**株式会社アスキー**
　〒107-24　東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
大代表　(03)3486-7111
出版営業部　(03)3486-1977 (ダイヤルイン)

制　作　東京書籍印刷株式会社
印　刷　東京書籍印刷株式会社

編　集　MSX マガジン編集部

ISBN4-7561-0621-8　　**Printed in Japan**

• 11619

ISBN4-7561-0621-8 C3055 P2500E

定価2,500円(本体2,427円)